Parallel Algorithms for Multiway Merging, Sorting and Graph Colouring

by

SAJITH G.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1997

Parallel Algorithms for Multiway Merging, Sorting and Graph Colouring

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

Doctor of Philosophy

by

SAJITH G.

to the

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

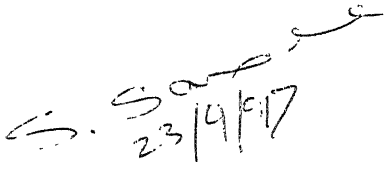
April 1997.

CSE-1997-D-SAJ-PAR

Entered in system 10-7-97

CERTIFICATE

It is certified that the work contained in the thesis entitled **Parallel Algorithms for Multiway Merging, Sorting and Graph Colouring** by **Sajith G.**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



Dr. Sanjeev Saxena
Associate Professor
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur.

April, 1997

Synopsis

This thesis concerns design and analysis of parallel algorithms on PRAMs and the parallel comparison model. The following problems are investigated:

1. Comparison-based merge sorting
2. Multiway merging
3. Optimal sublogarithmic time 3-vertex-colouring of rooted forests
4. 3-vertex-colouring of 4-clique free 3-degree graphs
5. Vertex-colouring of interval graphs
6. Edge-colouring of graphs

The **multiway merging** problem, where k non-empty sorted arrays of a total size n have to be merged into a single array, is studied as a generalisation of sorting and merging; when $k = 2$ we have merging and when $k = n$ we have sorting.

For **comparison-based multiway merging**, the existing CREW PRAM upper bound of $O(\log n)$ [97] is improved in two ways:

- an $O(\log k + \log\log n)$ time CREW PRAM optimal algorithm is obtained, and
- an $O(\log n)$ time EREW PRAM optimal algorithm is obtained.

While the former improves the time bound, the latter weakens the model from CREW to EREW. For both of these results, matching lower bounds are also obtained. For the CRCW PRAM model, an algorithm that runs in

$$O\left(\log\frac{\log n}{\log r} + \frac{\log k}{\log r}\cdot(\log\log r)^5\cdot 2^{O(\log^* k-\log^* r+1)} + \frac{\log k}{\log\log(kr)}\right)$$

time, with nr processors, $2 \le r \le n$, is presented. Also, it is shown that the lower bound on time for comparison-based k-way merging, with nr processors on a CRCW

PRAM, is

$$\Omega\left(\log\frac{\log n}{\log r}+\frac{\log k}{\log r}+\frac{\log k}{\log\log(kr)}\right).$$

Combining all these results, the parallel time complexity of comparison-based k-way merging is

$$\Theta(\textsc{Merge}(n,nr)+\textsc{Sort}(k,kr))$$

on all of EREW, CREW and CRCW PRAM models, with nr processors; here $\textsc{Merge}(n,p)$ (resp. $\textsc{Sort}(n,p)$) denotes the parallel time complexity of 2-way merging (resp. sorting) n items with p processors.

Multiway merging of integers is also studied. For k-way merging n integers drawn from the range $[0\ldots m-1]$, an ARBITRARY CRCW PRAM algorithm is given; this algorithm, with p processors, runs in

$$O\left(\frac{\log k}{\log\log k}+\log\log(\min\{n,\log m\})+\min\left\{\log k+\frac{n\log\log k}{p},\log\log mn+\frac{n\log\log m}{p}\right\}\right)$$

time. It is also shown that when the integers are from the range $[0\ldots\frac{n}{k}-1]$, k-way merging can be done in

$$O\left(\frac{n}{p}+\alpha(n/k)+\frac{\log k}{\log\log(pk/n)}+\log\frac{\log n}{\log(pk/n)}\right)$$

time, with p processors.

When the input items are single bit integers, the lower bound on time required for k-way merging, with p processors, on a CRCW PRAM, is shown to be $\Omega(\frac{\log k}{\log\log p})$; this lower bound is tight. However, when the integers are from the range $[0\ldots\frac{n}{k}-1]$, $\Omega(\frac{\log k}{\log\log(pk/n)})$ is a lower bound.

With the improved parallel multiway merging routine as the basic building block, a new **comparison-based parallel merge sorting** algorithm is obtained. This algorithm has small constant factors, and is very different from Cole's merge sort [19] in that it does not use a pipelining scheme. On the parallel comparison model, with nr processors, $2\leq r\leq n$, the algorithm sorts n items in

$$\frac{\log n}{\log r}\cdot 2^{O(\log^*(\log n/\log r))}$$

time—that is, faster than Cole's merge sort except for very small values of r. On this model, $\Omega(\log n/\log r)$ is a lower bound [9] on time for sorting with nr processors; the

only upper bound (Alon, 1987) that matches this lower bound, being an adaptation of the AKS sorting network, has large constant factors. On a CREW PRAM, with n processors, the algorithm runs in $\log n \cdot 2^{O(\log^* n)}$ time. On a CRCW PRAM, with nr processors, the algorithm runs in $\frac{\log n}{\log\log r} \cdot 2^{O(\log^*(\log n/\log r))}$ time.

For the problem of **3-colouring a rooted forest**, an $O((\log\log n)\log^*(\log^* n))$ time, **optimal parallel algorithm** is presented; TOLERANT CRCW PRAM is the model used. For this problem, an $O(\log(\log^* n))$ time, n processors, suboptimal, CREW PRAM algorithm has been known for long [48]. But a sublogarithmic time, optimal parallel algorithm has not hitherto been known, even on a CRCW PRAM.

Furthermore, it is shown that if $f(n)$ is the running time of the best known algorithm for 3-colouring a rooted forest on a COMMON or TOLERANT CRCW PRAM, a fractional independent set of the rooted forest can be found in $O(f(n))$ time, with the same number of processors, on the same model.

Using these results, it is shown that decomposable top-down algebraic computation, and hence depth computation (ranking), 2-colouring and prefix summation on rooted forests can be done in $O(\log n)$ optimal time on a TOLERANT CRCW PRAM.

These algorithms have been obtained by proving a result of independent interest, one concerning the **self-simulation property of TOLERANT**: an N-processor TOLERANT CRCW PRAM that uses an address space of size $O(N)$ only, can be simulated on an n-processor TOLERANT PRAM in $O(\frac{N}{n})$ time, with no asymptotic increase in space or cost, when $n = O(N/\log\log N)$.

By **Brooks' theorem**, for $\Delta \geq 3$, any $(\Delta+1)$-clique-free Δ-degree graph (called a Brooks' graph) is Δ-vertex-colourable. An algorithm for Δ-colouring a Δ-degree Brooks' graph in $O(\Delta^2 \log\Delta \log n)$ time, with $n/\log n$ processors, on an EREW PRAM, is obtained. In particular, a 3-degree Brooks' graph is 3-coloured in $O(\log n)$ time, with $n/\log n$ processors, on an EREW PRAM. The basic idea of this algorithm, in the context of a CREW PRAM, was treated in the author's M. Tech. thesis.

Besides, the following combinatorial result is obtained: for any two vertices u and v that are more than a constant distance apart in a 3-degree Brooks' graph G, there exist two distinct 3-colourings of G of which one has u and v coloured the same, and the other has u and v coloured differently. Thus, no vertex in a Brooks' graph

can force the colour of vertices arbitrarily far away; in contrast, while 2-colouring a linked list, fixing the colour of one vertex fixes the colour of all others. This highly local nature of the problem can be seen as suggesting that on a CRCW PRAM, an $\Omega(\log n/\log\log n)$ time lower bound may not hold.

The complexity of **minimally colouring an interval graph that has a known interval representation** (let us denote this problem by IGC) on a bounded fan-in circuit is studied. The following results are obtained:

- The problem of **3-colouring a linked list** is NC^1-reducible to IGC.
- When the chromatic number of the input interval graph is restricted to at most a constant, IGC is in NC^1.

Using these two results, it is shown that a linked list that has at most a constant number of stretches, can be 3-coloured in NC^1; here a linked list is visualised as being constituted of alternating stretches of forward and backward pointers in an array. This complements the observation that 3-colouring a linked list, in general, is unlikely to be in NC^1 [72]. Note that in view of this observation, the NC^1-reduction from 3-colouring a list to IGC implies that IGC, in general, is unlikely to be in NC^1.

The complexity of IGC on a PRAM is also studied.

For interval graphs of $o(\log n)$ chromatic number, a $o(\log n)$ time, polynomial processors, CRCW PRAM algorithm is obtained. In particular, when the chromatic number is $O((\log n)^{1-\epsilon})$, $0 < \epsilon < 1$, the algorithm runs in $O(\log n/\log\log n)$ time. An $O(\log n)$ time, $O(n)$ cost, EREW PRAM algorithm is found for general interval graphs.

Complementing these algorithms, the following lower bound result is obtained: even when the left and right endpoints of the intervals are separately sorted, IGC needs $\Omega(\log n/\log\log n)$ time, on a CRCW PRAM, with a polynomial number of processors. This result assumes significance, because, the $\Omega(\log n/\log\log n)$ time lower bound [16] on finding the chromatic number χ of an interval graph is not valid when either the left and right endpoints are separately sorted, or χ is also taken as a parameter; in the former case χ can be found in $O(1)$ time, and in the latter in $O(\log\chi/\log\log n)$ time, with a polynomial number of processors.

By Vizing's theorem, any Δ-degree graph is $(\Delta+1)$-**edge-colourable.** But

$(\Delta + 1)$-edge-colouring of arbitrary graphs in parallel has proved to be difficult; NC algorithms are known only for graphs with Δ at most a polylogarithmic in n [60]. To obtain efficient parallel edge-colouring algorithms, thus, more colours may have to be allowed. Probing in this vein, the following EREW PRAM algorithms for edge-colouring a Δ-degree graph are obtained:

- an algorithm that finds a $(\Delta + d)$-edge-colouring, $1 \leq d < \Delta$, in $O((\log d + (\Delta/d)^4)\log^2 n)$ time, using $n + m$ processors
- an algorithm that finds a $\Delta^{1+\epsilon}$-edge-colouring, $0 < \epsilon < 1$, in $O(\log \Delta \log(\log^* n))$ time, using $n\Delta^{1+\epsilon}$ processors

Acknowledgments

I am extremely indebted to my thesis supervisor Dr. Sanjeev Saxena; without his patient and insistent guidance, this thesis would not have been possible. I thank the Dept. of Computer Science, IIT Kanpur, and the Govt. of India for making this research possible.

I am very thankful to Dr. Ghosh for encouraging me, and for his comments on this work.

Also, I am grateful to Dr. Karnick, Dr. Sangal, Dr. S. K. Aggarwal, Dr. Barua, Dr. Biswas, Dr. Ghosh, Dr. P. Gupta, and Dr. Moona, who have instructed me during the course of my studentship in this department.

I thank my colleagues in the department S. V. Rao, Deepak, Reddy, Naik, Suresh, Ms. Bansal, and Gore for their companionship.

Without my friends Apu and George, my long stay at IIT Kanpur would just not have been the same. I acknowledge all my friends, old and new, and particularly of Hall 5, who formed a great part of my world around here.

And above all, I owe this work to the loving support of my parents and sister; I dedicate this work to them.

Sajith G.

Contents

Chapter 1

Introduction

With the speed of the present day processors approaching the maximum physically possible, we can no longer expect technological innovations to provide significant improvements in the speed of single processor systems [51]. Meanwhile, with the increasing use of computers in complex computational tasks, the demand for higher performance in computing is also on the increase. To meet this demand, thus, it is necessary to concentrate on parallel processing. If we were to employ p processors, $p > 1$, concurrently in solving computational tasks, even though inherent sequentialities of the tasks may prevent an ideal speed up of $O(p)$, many times a non-trivial $\omega(1)$ speed up would still be achievable. Hence, it is to be expected that in near future, general purpose computers may well be parallel processors. Hence the research interest in parallel processors and algorithms.

Over the recent years, a large body of knowledge on design and analysis of parallel algorithms has developed. This thesis is an effort to contribute to the same.

1.1 Models of Parallel Computation

The concerns of parallel algorithm design are often very different from those of sequential algorithm design. The performance of a parallel algorithm depends on a variety of factors like processor allocation, synchronisation and resource sharing, which are not present in sequential computation. Because of these, a consensus has not yet been reached on the ideal model of computation to be used in designing parallel algorithms. The Parallel Random Access Machine (PRAM) is probably the most widely

used model of parallel computation; in this thesis, we shall be mostly concerned with designing algorithms on this model.

A PRAM [61, 55] has p synchronous processors, all having access to a common memory. Each processor is similar to a random access machine, the standard model of sequential algorithm design [2], and is assigned a unique index from the range $[1 \ldots p]$. PRAM models can be classified according to the constraints they impose on the global memory access. An Exclusive Read Exclusive Write (EREW) PRAM does not allow simultaneous access by more than one processor to the same memory location, for read or write purposes. A Concurrent Read Exclusive Write (CREW) PRAM allows simultaneous access for reads, but not for writes. A Concurrent Read Concurrent Write (CRCW) PRAM allows simultaneous access for both reads and writes. Based on the write conflict resolution schemes used, CRCW PRAMs can be further sub-classified.
PRIORITY CRCW PRAM: in any concurrent write, the processor with the lowest index succeeds.
ARBITRARY CRCW PRAM: in any concurrent write, an arbitrary one of the processors succeeds; the programs should work irrespective of the identity of the successful processor.
COMMON CRCW PRAM: all processors involved in a concurrent write should write the same value.
COLLISION CRCW PRAM: when multiple processors write in the same memory location a special collision symbol appears in that location.
TOLERANT CRCW PRAM: when multiple processors write in the same memory location, the content of that location remains unchanged.

A PRAM model is said to be *self-simulating* , if for all $N \geq n \geq 1$, a PRAM of that model of size n can simulate a single step of another PRAM of the same model of size N in $O(\frac{N}{n})$ time. All CRCW PRAMs that are at least as powerful as COLLISION or COMMON are self-simulating. TOLERANT is not known to be self-simulating [39] (see also [4]).

In Chapter 4, we show that a TOLERANT PRAM of size N with a linear address space, can be slowed down by any factor $\lambda = \Omega(\log \log N)$, with no asymptotic increase in space or cost.

The cost or work of an algorithm that runs in t time using p processors i
time-processor prc

If Seq(n) is the worst-case running time of the fastest known sequential algorithm for a problem of size n, an optimal parallel algorithm for the same problem runs in $O(\text{Seq}(n)/p)$ time using p processors.

Solutions for some algorithmic problems like sorting and merging are typically dominated by comparisons between items drawn from a linearly ordered set. For these problems, it is useful and instructive to design algorithms on the parallel comparison model [96]. In this model, there are p processors synchronised by a common clock. In each step of the clock, a processor can compare two items and find their relative ordering. Thus, in each step of an algorithm, comparisons are done between at most p pairs of (not necessarily distinct) items. The algorithm incurs no cost in using the results of prior comparisons to decide which comparisons to make next. As soon as enough ordering relations have been found to formulate the output as required, the algorithm terminates. The number of steps executed till then is the running time of the algorithm. This model is clearly more powerful than all standard PRAM models.

Another model helpful in studying comparison intensive problems on items from a linearly ordered set is the comparator network. A comparator is a two-input module with an ordered pair of outputs: if a and b are the inputs, then the first output is $\min\{a, b\}$, and the second output is $\max\{a, b\}$. A comparator network is made only of comparators; its size is the number of comparators in it, and its depth is the length of the longest path in it from an input to an output. This model is clearly weaker than all standard PRAM models.

A boolean circuit [61] is a directed acyclic graph in which each node is an input, or an output, or a 1, or a 0, or an AND gate, or an OR gate, or a NOT gate. For a boolean circuit, the size is the number of edges in it, and the depth is the length of the longest path in the circuit from an input to an output. When the in-degree (fan-in) of a node is restricted to a constant the circuit is said to have a bounded fan-in, otherwise an unbounded fan-in. NC^k, $k \geq 0$ is the class of all problems solvable in $O(\log^k n)$ depth on a bounded fan-in circuit of a polynomial size. NC is the class $\cup_{k \geq 0} \text{NC}^k$. AC^k, $k \geq 0$ is the class of all problems solvable in $O(\log^k n)$ depth on an unbounded fan-in circuit of a polynomial size. AC is the class $\cup_{k \geq 0} \text{AC}^k$. It is known that NC = AC. Moreover,

$$\text{NC}^k \subseteq \text{EREW}^k \subseteq \text{CREW}^k \subseteq \text{CRCW}^k \subseteq \text{AC}^k \subseteq \text{NC}^{k+1}$$

FL [24] is the class of problems solvable in deterministic log space; here a

problem is a multiple valued function and solving a problem involves finding any one of its possible values for a given input. It is known that $NC^1 \subseteq FL$, but whether the inclusion is proper or not is still not known.

1.2 Sorting, Merging and Graph Colouring

This thesis is mainly concerned with sorting, merging and graph colouring.

Sorting is a fundamental tool for algorithm designers. A large percentage of computer time is typically spent in sorting [9, 64]. So, this problem has been, and still is, intensely researched. While dealing with sorted lists, time and again, it becomes necessary to merge several into one. Conversely, merging has been used as a basic tool in designing sorting algorithms: merge sort has been one of the first $O(n \log n)$ time comparison-based sequential sorting algorithm to be discovered, whereas Cole's merge sort is the only optimal and yet simple comparison-based parallel sorting algorithm.

Graph colouring assumes special significance in the parallel world because of its applications in symmetry breaking and scheduling. For example, a context, where a number of entities vie for a number of shared resources, can be captured in a conflict graph, and a colouring of this graph can give a schedule for resource allocation.

1.2.1 Sorting and Merging

Sorting is the problem of arranging a set of elements drawn from a linearly ordered set in nondecreasing or non-increasing order. Merging, on the other hand, is combining two sorted lists into a single one. In the multiway merging problem, we are given k sorted arrays, and we have to merge them to get a single sorted array. When the value of k is understood, we shall also call this problem k-way merging. When $k = 2$, k-way merging degenerates into ordinary merging. The special case of $k = n$, where n is the total number of elements, is sorting.

When the input elements are considered atomic entities, comparison being the only operation performable on them, sorting, 2-way merging and multiway merging are said to be comparison-based.

Sorting n elements, using only comparisons, requires $\Omega(n \log n)$ time sequentially, and there are several algorithms that match this lower bound [2, 64]. In the parallel setting, Batcher gave $O(\log^2 n)$ depth, $O(n \log^2 n)$ cost comparator networks

for sorting [64]. Finding a logarithmic depth network for sorting remained an open question for long, until Ajtai, Komlós and Szemerédi [3, 81, 82] found the $O(\log n)$ depth, $O(n \log n)$ size, AKS sorting network; though this network achieves an $O(\log n)$ depth, the constant factor hidden by the O-notation here is quite large. Cook, Dwork and Reischuk [23] showed that on a CREW PRAM, finding the OR of n bits requires $\Omega(\log n)$ time, with any number of processors; the same lower bound holds for sorting as well. So, a simulation of the AKS sorting network is optimal on all models that are not more powerful than the CREW PRAM model. Cole gave a simple, logarithmic time, linear processors, EREW PRAM sorting algorithm that has small constant factors [19]. For the parallel comparison model, Azar and Vishkin [9] have shown that, sorting n elements with $p \geq n$ processors requires at least $\Omega(\frac{\log n}{\log(p/n)})$ comparison steps. The only upper bound (due to Alon, [9]) that matches this lower bound, being an adaptation of the AKS sorting network, has large constant factors. Cole's merge sort runs on the parallel comparison model in $O(\frac{\log n}{\log(p/n)} \cdot \log\log(p/n))$ steps. For the CRCW PRAM model, Beame and Hastad [10] proved that computing the parity of n bits requires $\Omega(\frac{\log n}{\log\log p})$ time with $p \geq n$ processors. Combining this result with the parallel comparison model lower bound, sorting n elements on a CRCW PRAM with $p \geq n$ processors requires $\Omega(\frac{\log n}{\log(p/n)} + \frac{\log n}{\log\log p})$ time, because, parity is reducible to sorting [10]. No algorithm matching this lower bound is known today. The fastest known algorithm takes

$$O\left(\frac{\log n}{\log(p/n)}(\log\log(p/n))^5 2^{O(\log^* n - \log^*(p/n)+1)} + \frac{\log n}{\log\log p}\right)$$

time with $p \geq n$ processors [45]. Cole's merge sort runs on a CRCW PRAM in $O(\frac{\log n}{\log\log(p/n)})$ time.

Sorting algorithms have been designed on other models of parallel computation too. In particular, on the parallel comparison model, Columnsort of Leighton [68] runs in $O((\log n/\log r)^{3.419})$ steps, and Cubesort of Cypher and Sanz [25] runs in $O(2^{\log^*(n/r)}(\log n/\log r)^2)$ steps, with $2 \leq r \leq n$ processors.

Comparison based 2-way merging, in contrast, takes only linear time sequentially. In the parallel setting, Batcher gave merging networks of $O(\log n)$ depth and $O(n \log n)$ size [64]. These networks are asymptotically the best possible, because, any merging network must have $\Omega(n \log n)$ comparator modules [64]. However, on an EREW PRAM we can do better: Hagerup and Rüb [46] describe how to obtain an $O(\log n)$ time optimal algorithm using Batcher's networks. Also, for the EREW

PRAM model, a reduction from searching to 2-way merging, along with the logarithmic time lower bound for searching [93], implies that 2-way merging requires $\Omega(\log n)$ time with any number of processors. Borodin and Hopcroft [14] showed that on the parallel comparison model, with $p \geq n$ processors, $\Omega(\log \frac{\log n}{\log(p/n)})$ time is required to merge two sorted arrays of a total size n. This matches the best known upper bound on the CREW PRAM model [14, 96]. From this suboptimal CREW PRAM algorithm, Kruskal has obtained an optimal $O(\log \log n)$ time CREW PRAM algorithm [65].

A decision tree argument shows that k-way merging by comparisons requires $\Omega(n \log k)$ time sequentially [97]. Also, using a heap for finding the minimum of k values, k-way merging can be done in $O(n \log k)$ sequential time [64]. Lee and Batcher [67] have generalised the odd-even merge principle to find an $O(\log n \log k)$ depth, k-way merging network. Wen [97] has obtained an $O(\log n)$ time optimal parallel algorithm on a CREW PRAM.

If the input elements are assumed to be integers, the lower bound of $\Omega(n \log n)$ is no longer valid for sorting. The best known upper bound is $O(n \log \log n)$ [7], whereas no stricter lower bound than the trivial $\Omega(n)$ is known. Parallel integer sorting on a CRCW PRAM must obey the $\Omega(\frac{\log n}{\log \log p})$ time lower bound of Beame and Hastad, because of the reduction from parity. Andersson et al. [7] showed that on an ARBITRARY CRCW PRAM, n integers can be sorted in $O(\log n)$ time, with $O(n \log \log n)$ operations. Bhatt et al. [12] gave an $O(\frac{\log n}{\log \log n} + \log \log m)$ time, $O(n \log \log m)$ operations algorithm, for sorting integers in the range $[0 \ldots m-1]$, on an ARBITRARY CRCW PRAM.

For 2-way integer merging, Berkman and Vishkin [78] gave an $O(\log \log \log m)$ time optimal CREW PRAM algorithm for the case where the integers are from the range $[0 \ldots m-1]$. For the case of $m = n$, they gave an $O(\alpha(n))$ time optimal algorithm for the CRCW PRAM model, where α is the inverse Ackermann function. Hagerup and Kutylowski [43] described an $O(\log \log n + \log \min\{n, m\})$ time, optimal cost EREW PRAM algorithm for merging integers from the range $[0 \ldots m-1]$.

In Chapters 2 and 3 of this thesis, the following results on sorting and multiway merging are presented.

For comparison-based multiway merging, the existing CREW PRAM upper bound of $O(\log n)$ [97] is improved in two ways:

- an $O(\log k + \log\log n)$ time CREW PRAM optimal algorithm is obtained, and
- an $O(\log n)$ time EREW PRAM optimal algorithm is obtained.

While the former improves the time bound, the latter weakens the model from CREW to EREW. For both of these results, matching lower bounds are also obtained. For the CRCW PRAM model, an algorithm that runs in

$$O\left(\log\frac{\log n}{\log r} + \frac{\log k}{\log r}\cdot(\log\log r)^5\cdot 2^{O(\log^* k-\log^* r+1)} + \frac{\log k}{\log\log(kr)}\right)$$

time, with nr processors, $2 \le r \le n$, is presented. Also, it is shown that the lower bound on time for comparison-based k-way merging, with nr processors on a CRCW PRAM, is

$$\Omega\left(\log\frac{\log n}{\log r} + \frac{\log k}{\log r} + \frac{\log k}{\log\log(kr)}\right).$$

Combining all these results, the parallel time complexity of comparison-based k-way merging, with nr processors, is

$$\Theta(\textsc{Merge}(n, nr) + \textsc{Sort}(k, kr))$$

on all of EREW, CREW and CRCW PRAM models; here $\textsc{Merge}(n,p)$ (resp. $\textsc{Sort}(n,p)$) denotes the parallel time complexity of 2-way merging (resp. sorting) n items with p processors.

Multiway merging of integers is also studied. For k-way merging n integers drawn from the range $[0\ldots m-1]$, an ARBITRARY CRCW PRAM algorithm is given; this algorithm, with p processors, runs in

$$O\left(\frac{\log k}{\log\log k} + \log\log(\min\{n, \log m\}) + \min\left\{\log k + \frac{n\log\log k}{p}, \log\log mn + \frac{n\log\log m}{p}\right\}\right)$$

time. It is also shown that when the integers are from the range $[0\ldots\frac{n}{k}-1]$, k-way merging can be done in

$$O\left(\frac{n}{p} + \alpha(n/k) + \frac{\log k}{\log\log(pk/n)} + \log\frac{\log n}{\log(pk/n)}\right)$$

time, with p processors.

When the input items are single bit integers, the lower bound on time required for k-way merging, with p processors, on a CRCW PRAM, is shown to be $\Omega(\frac{\log k}{\log\log p})$; this lower bound is tight. However, when the integers are from the range $[0\ldots\frac{n}{k}-1]$, $\Omega(\frac{\log k}{\log\log(pk/n)})$ is a lower bound.

Using the improved parallel multiway merging algorithm as the basic building block, a new comparison-based parallel merge sorting algorithm is obtained. This is presented in Chapter 3. This algorithm has small constant factors, and is very different from Cole's merge sort [19] in that it does not use a pipelining scheme. On the parallel comparison model, with nr processors, $2 \leq r \leq n$, the algorithm sorts n items in

$$\frac{\log n}{\log r} \cdot 2^{O(\log^*(\log n/\log r))}$$

time—that is, faster than Cole's merge sort except for very small values of r. On a CREW PRAM, with n processors, the algorithm runs in $\log n \cdot 2^{O(\log^* n)}$ time. On a CRCW PRAM, with nr processors, the algorithm runs in $\frac{\log n}{\log\log r} \cdot 2^{O(\log^*(\log n/\log r))}$ time.

1.2.2 Graph Theoretic Terms and Notations

We follow the graph theoretic notations of Harary [50] and Golumbic [36].

A graph $G = (V, E)$ consists of a finite nonempty set V of vertices and a set $E \subseteq V \times V$ of unordered pairs of distinct vertices [50]. Each $e \in E$ is an edge of G. Throughout this thesis, we shall denote the number of edges $|E|$ by m, the number of vertices $|V|$ by n. Two vertices $u, v \in V$ are said to be adjacent iff $\{u, v\} \in E$; in this case, we also say that u and v are neighbours of each other. We denote the set of neighbours of v by $N(v)$. When $e = \{u, v\}$, we say that e is incident with u and v. The number of neighbours of a vertex is its degree. The maximum vertex degree of a graph G is denoted by $\Delta(G)$; whenever there is only one graph G under consideration we shall use Δ for $\Delta(G)$. A graph in which every vertex is of degree d, is a d-regular graph. A 3-regular graph is also called a cubic graph. A graph that has a maximum vertex degree of 3 is called a subcubic graph.

In a directed graph every edge is an ordered pair; if $(u, v) \in E$ we say that v is an out-neighbour of u and u is an in-neighbour of v. The number of out-neighbours of a vertex is its out-degree, and the number of in-neighbours is its in-degree.

$G' = (V', E')$ is said to be a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$. If $E' = \{\{u, v\} \mid \{u, v\} \in E \text{ and } u, v \in V'\}$ then, $G' = G[V']$ is the subgraph induced by V'. We use $G - U$ to denote $G[V - U]$ for $U \subseteq V$, $G - v$ to denote $G - \{v\}$ for $v \in V$, and $G - e$ to denote $G' = (V, E - \{e\})$ for $e \in E$.

A graph is called complete if for every pair (u, v) of distinct vertices, u and v are adjacent. A complete graph on n vertices is called an n-clique. A maximum clique of G is a largest cardinality complete subgraph of G; its cardinality is the clique number $\omega(G)$ of G.

An independent set I of G is a subset of V such that for every $u, v \in I$, $\{u, v\}$ is not in E. In a maximal independent set (MIS for short) I, in addition the following condition holds: for each $v \in V$ either $v \in I$ or v has a neighbour in I.

In the vertex colouring problem on graphs, we seek to assign colours to the vertices so that no two adjacent vertices get the same colour. The minimum number of colours needed to colour G is the chromatic number of G and is denoted by $\chi(G)$. Clearly, $\chi(G)$ is not smaller than $\omega(G)$, the clique number. If, for every $V' \subseteq V$, $\chi(G[V']) = \omega(G[V'])$, then $G = (V, E)$ is called a perfect graph.

A graph $G = (V, E)$ is called an interval graph, if for some set $\Im$ of intervals of a linearly ordered set, there is a bijection $f : V \rightarrow \Im$ so that two vertices u and v are adjacent in G iff $f(u)$ and $f(v)$ overlap. An interval graph is known to be perfect [36].

A bipartite graph G is a graph whose vertex set V can be partitioned into two subsets V_1 and V_2 so that every edge in G has one endvertex in V_1 and the other endvertex in V_2. A bipartite graph is 2 vertex-colourable. Any graph without cycles is a forest. Each connected component of a forest is a tree. A rooted forest $F = (V, E)$ is a directed forest in which the out-degree of each vertex is at most one. Roots in F are the vertices with out-degree exactly zero. Each edge in E is from a child to its parent; that is, if $(u, v) \in E$, v is the parent of u, and u is a child of v. Forests are bipartite graphs.

In the edge-colouring problem on graphs, we seek to assign colours to the edges so that no two edges incident with the same vertex get the same colour. For a graph $G = (V, E)$, the minimum number of colours required to edge-colour G is the chromatic index $\chi'(G)$ of G. Clearly, $\chi' \geq \Delta$.

1.2.3 Vertex Colouring

Finding the chromatic number of a 4-degree planar graph, let alone an arbitrary graph, is NP complete [29]. Minimally vertex colouring an arbitrary graph is, thus, computationally difficult.

Hence, to colour graphs efficiently we should either restrict our attention to

certain types of graphs that are easy to colour minimally, or settle for a possibly non-minimal colouring.

In the sequential setting, both these approaches have given efficient algorithms. A simple $O(m)$ time search can 2-colour a bipartite graph. The proof of 4-colour theorem gives a polynomial time algorithm for 4-colouring a planar graph; this algorithm has large constant factors. But planar graphs can be 5-coloured in $O(n)$ time: see [42] for a survey of such algorithms. Several special classes of perfect graphs have efficient algorithms for minimal colouring, though no polynomial time algorithm for recognising general perfect graphs is known. For example an interval graph can be recognised and minimally coloured in $O(n)$ time. A graph can be easily $(\Delta+1)$-coloured in $O(m)$ time. Brooks showed that any connected graph is Δ-colourable if it is neither a complete graph on $\Delta+1$ vertices nor a circuit of odd length [99, 73]. A Δ-colouring has often been called a *Brooks' colouring* and a Δ-colourable graph a *Brooks' graph.* A Brooks' graph can be Δ-coloured in linear time (see [73], problems 9.12 and 9.13).

In the parallel setting, a bipartite graph can be 2-coloured in $O(\log n)$ time. In particular, a linked list can be 2-coloured in $O(\log n)$ time optimally on an EREW PRAM [6]. A rooted forest can be 3-coloured in $O(\log(\log^* n))$ time with n processors on an EREW PRAM [48] Planar graphs can be 5-coloured in $O(\log n \log^* n)$ time and $O(n)$ work on an EREW PRAM [42]. Given an interval graph in adjacency list form, an interval representation for it can be found in $O(\log^2 n)$ time using $n+m$ processors on an EREW PRAM [63]. Given an interval representation, the interval graph can be minimally coloured in $O(\log n)$ time with n processors on an EREW PRAM [89].

Closely associated to $(\Delta+1)$-colouring is the MIS problem; this is because, there is an NC reduction from $(\Delta+1)$-colouring to the MIS problem [74]; thus an NC-algorithm for the MIS problem implies an NC-algorithm for $(\Delta+1)$-colouring as well. Karp and Wigderson [62] describe an $O((\log n)^4)$ time algorithm for the MIS problem, with $O((\frac{n}{\log n})^3)$ processors on an EREW PRAM, thus proving the problem to be in NC. Alon, Babai and Itai [5] and Luby [74] have obtained fast and simple randomised algorithms for the MIS problem; these algorithms take $O((\log n)^2)$ expected time with $O(m)$ processors on an EREW PRAM. Alon et.al. [5] and Luby [74] also present derandomisation techniques to obtain fast deterministic versions of their respective algorithms. The first linear processors NC algorithm for the MIS problem is due to Goldberg and Spencer [34], and takes $O((\log n)^4)$ time on an EREW PRAM; they later improved

the time bound to $O((\log n)^3)$ [33]. Luby [75] describes a derandomisation technique without processor penalty; this results in a linear processors $O((\log n)^3 \log\log n)$ time algorithm for $(\Delta+1)$-colouring on a CREW PRAM.

For the special case of Δ being bounded, [32] gives an $O(n)$ processors $O(\log^* n)$ time algorithm for the EREW PRAM model whereas an optimal algorithm that takes $O(\log^{(k)} n)$ time with $\frac{n}{\log^{(k)} n}$ processors, again on an EREW PRAM, is given in [87]; for general graphs these algorithms take, respectively, $O(\Delta \log \Delta(\log^* n + \Delta))$ time and $O(\Delta^2 \log \Delta \log^{(k)} n)$ time.

The problem of Brooks' colouring, is known to be in NC [57, 59, 47, 80]. Panconesi and Srinivasan [80] prove a $\Theta(\log_\Delta n)$ bound on the search radius required to fix the colour of a lone uncoloured node of a Brooks' graph. Using this result, they obtain an $O(\frac{\log^3 n}{\log \Delta})$ time reduction from Brooks' colouring to to $(\Delta+1)$-colouring [80]. On bounded degree graphs their algorithm can be implemented with $O(n)$ processors in $O(\log^2 n \log^* n)$ time on an EREW PRAM.

For Brooks' colouring of cubic graphs (i.e., $\Delta = 3$), Karloff [58] give an algorithm that can be implemented in $O(\log^2 n)$ time with $\frac{n}{\log n}$ processors on an EREW PRAM. Furthermore, Karloff [58] shows that Brooks' colouring an α-degree graph G can be reduced in NC, to Brooks' colouring an $(\alpha-1)$-degree subgraph of G, for $\alpha > 3$. This reduction requires a spanning tree in an $O(n)$-vertex graph to be found, and so, even for bounded degree graphs, can at best be implemented in $O(\log n)$ time with $\frac{(n+m)\alpha(m,n)}{\log n}$ processors on a CRCW PRAM [21, 54] and in $O(\log n \log\log n)$ time with n processors on an EREW PRAM [17]. So, on bounded degree graphs, the consequent Brooks' colouring algorithm, in [58] takes $O(\log^2 n)$ time with n processors on an EREW PRAM and $O(\log^2 n)$ time with $\frac{(n+m)\alpha(m,n)}{\log n}$ processors on a CRCW PRAM.

In Chapters 4, 5 and 6 of this thesis, the following results related to vertex colouring are obtained.

For the problem of 3-colouring a rooted forest, an $O((\log\log n)\log^*(\log^* n))$ time, optimal parallel algorithm is presented, in Chapter 4; TOLERANT CRCW PRAM is the model used. For this problem, a sublogarithmic time, optimal parallel algorithm has not hitherto been known, even on a CRCW PRAM. Furthermore, it is shown that if $f(n)$ is the running time of the best known algorithm for 3-colouring a rooted forest on a COMMON or TOLERANT CRCW PRAM, a fractional independent

set of the rooted forest can be found in $O(f(n))$ time, with the same number of processors, on the same model. Using these results, it is shown that decomposable top-down algebraic computation, and hence depth computation (ranking), 2-colouring and prefix summation on rooted forests can be done in $O(\log n)$ optimal time on a TOLERANT CRCW PRAM.

In Chapter 5, we study Brooks' colouring. An algorithm for Δ-colouring a Δ-degree Brooks' graph in $O(\Delta^2 \log \Delta \log n)$ time, with $n/\log n$ processors, on an EREW PRAM, is obtained. In particular, this algorithm 3-colours a 3-degree Brooks' graph in $O(\log n)$ time, with $n/\log n$ processors, on an EREW PRAM.

Besides, the following combinatorial result is obtained: for any two vertices u and v that are more than a constant distance apart in a 3-degree Brooks' graph G, there exist two distinct 3-colourings of G of which one has u and v coloured the same, and the other has u and v coloured differently. Thus, no vertex in a Brooks' graph can force the colour of vertices arbitrarily far away; in contrast, while 2-colouring a linked list, fixing the colour of one vertex fixes the colour of all others. This highly local nature of the problem can be seen as suggesting that on a CRCW PRAM, an $\Omega(\log n/\log\log n)$ time lower bound may not hold.

In Chapter 6, the complexity of minimally colouring an interval graph that has a known interval representation (let us denote this problem by IGC) on a bounded fan-in circuit is studied. The following results are obtained:

- The problem of 3-colouring a linked list is NC^1-reducible to IGC.
- When the chromatic number of the input interval graph is restricted to at most a constant, IGC is in NC^1.

Using these two results, it is shown that a linked list that has at most a constant number of stretches, can be 3-coloured in NC^1; here a linked list is visualised as being constituted of alternating stretches of forward and backward pointers in an array. This complements the observation that 3-colouring a linked list, in general, is unlikely to be in NC^1 [72]. Note that in view of this observation, the NC^1-reduction from 3-colouring a list to IGC implies that IGC, in general, is unlikely to be in NC^1.

The complexity of IGC on a PRAM is also studied.

For interval graphs of $o(\log n)$ chromatic number, a $o(\log n)$ time, polynomial processors, CRCW PRAM algorithm is obtained. In particular, when the chromatic

number is $O((\log n)^{1-\epsilon})$, $0 < \epsilon < 1$, the algorithm runs in $O(\log n / \log\log n)$ time. An $O(\log n)$ time, $O(n)$ cost, EREW PRAM algorithm is found for general interval graphs.

Complementing these algorithms, the following lower bound result is obtained: even when the left and right endpoints of the intervals are separately sorted, IGC needs $\Omega(\log n / \log\log n)$ time, on a CRCW PRAM, with a polynomial number of processors. This result assumes significance, because, the $\Omega(\log n / \log\log n)$ time lower bound [16] on finding the chromatic number χ of an interval graph is not valid when either the left and right endpoints are separately sorted, or χ is also taken as a parameter; in the former case χ can be found in $O(1)$ time, and in the latter in $O(\log\chi / \log\log n)$ time, with a polynomial number of processors.

1.2.4 Edge Colouring

The chromatic index χ' of a graph G is at least $\Delta(G)$. Vizing [26, 13, 76, 56] and Gupta [56] showed that $\chi' \leq \Delta + 1$; this result is popularly known as Vizing's theorem. Thus, the chromatic index of a graph is either Δ or $\Delta + 1$; accordingly, graphs are said to be in either Class 1 or Class 2. Deciding the class of a graph is known as the classification problem. Even when restricted to cubic graphs, the classification problem is NP-complete [52]. Optimally edge colouring an arbitrary graph is, thus, computationally difficult.

To get efficient edge-colouring algorithms, hence, we should either restrict our attention to certain types of graphs that are easy to edge-colour optimally, or settle for a possibly nonoptimal $(\Delta+1)$-edge-colouring of general graphs.

On the sequential front, both approaches have given efficient algorithms. Planar graphs with $\Delta \geq 8$ and bipartite graphs are in Class 1 [26]. A planar graph with $\Delta \geq 8$ can be Δ-edge-coloured in $O(n^2)$ time (the result is quoted in [18]), whereas, a bipartite graph can be Δ-edge-coloured in $O(\min\{m \log n, n^2 \log n\})$ time [20, 28]. An arbitrary graph can be $(\Delta+1)$-edge-coloured in $O(\min\{mn, m\Delta \log n, m\sqrt{n \log n}\})$ (the result is quoted in [18]).

In the parallel setting, NC algorithms have been obtained for Δ-edge-colouring planar graphs with large Δ, as well as bipartite graphs. Planar graphs with $\Delta \geq 19$, can be Δ-edge-coloured in $O(\log^2 n)$ time on an EREW PRAM with n processors [18], while planar graphs with $\Delta \geq 8$, can be Δ-edge-coloured in $O(\log^3 n)$ time on an EREW PRAM with n^2 processors [18]. A bipartite graph can be Δ-edge-coloured in

$O(\log^2 n \log \Delta)$ time on an EREW PRAM with $n+m$ processors [69], and $2^{\lceil \log \Delta \rceil}$-edge-coloured in $O(\log n \log \Delta)$ time on an EREW PRAM with $n+m$ processors [69].

In contrast, $(\Delta+1)$-edge-colouring of arbitrary graphs is not yet known to be in NC; Karloff and Shmoys [60] give an algorithm that is in NC only when Δ is at most polylogarithmic in n. This algorithm runs in

$$O(\Delta^5 \log n(\Delta + \log n + \min\{\log^3 n, \Delta^2 \log \Delta(\log^* n + \Delta^2)\}))$$

time on an EREW PRAM with $n+m$ processors. Liang, Shen and Hu [71, 70] give an $O(\Delta^{4.5} \log^3 \Delta \log n + \Delta^4 \log^4 n)$ time, $n^3 \log \Delta + n\Delta^3$ processors CRCW PRAM algorithm for $(\Delta+1)$-edge-colouring a general graph.

To edge-colour an arbitrary graph in NC, thus, we may have to use more than $\Delta + 1$ colours. Fürer and Raghavachari's results [27] are in this vein. They give an $O(\log n \log \Delta)$ time, $n+m$ processors, CREW PRAM algorithm for $c\Delta$-edge-colouring a general graph, $c > 1$, and an $O(\log^* n)$ time, $n+m$ processors, CREW PRAM algorithm for Δ^2-edge-colouring a general graph.

In Chapter 8 of this thesis, the following EREW PRAM algorithms for edge-colouring a general Δ-degree graph are presented:

- an algorithm that finds a $(\Delta + d)$-edge-colouring, $1 \leq d < \Delta$, in $O((\log d + (\Delta/d)^4) \log^2 n)$ time, using $n+m$ processors
- an algorithm that finds a $\Delta^{1+\epsilon}$-edge-colouring, $0 < \epsilon < 1$, in $O(\log \Delta \log(\log^* n))$ time, using $n\Delta^{1+\epsilon}$ processors

Chapter 2

Multiway Merging in Parallel

Multiway merging is the problem of merging k sorted arrays into a single sorted array. When the value of k is understood, we shall also call this problem k-way merging. This problem can be seen as a generalisation of sorting and merging: the special case of $k = 2$ is ordinary merging, and that of $k = n$ is sorting.

Parallel solutions for the multiway merging problem form the topic of this chapter. An algorithm schema for multiway merging is described in Section 2.1. The implementation of the schema on various PRAM models for the comparison based version of the problem is discussed in Section 2.3; matching lower bounds are also presented. These results improve the existing CREW PRAM upper bound of $O(\log n)$ [97]. Multiway merging of integers is considered in Section 2.4.

2.1 The Algorithm Schema

The following definitions are due to Cole [19]. For three keys x, y and z, we say that y is *between* x and z, or that x and z *straddle* y, if $x \leq y < z$. For a key x and a list L, the *rank* of x in L is the number of elements in L that are not larger than x. A list L_1 is said to be *ranked* in another list L_2 if every element of L_1 knows its rank in L_2; L_1 and L_2 are *cross ranked* if they are ranked into each other.
The algorithm schema is now described:

Input: Arrays A_i of size n_i for $1 \leq i \leq k$, each sorted in increasing order, where $\sum_{i=1}^{k} n_i = n$; p processors are available. Without loss of generality, we assume that all elements are distinct, because otherwise, for each i, the j-th element a_j^i of A_i, can be

replaced by a triplet $\langle a_j^i, i, j\rangle$, with the linear ordering extended lexicographically.

Output: $A_1 \bigcup \ldots \bigcup A_k$ sorted in increasing order in an array $B[1 \ldots n]$.

Step 1: For a parameter λ to be appropriately chosen, if $n < k\lambda$, sort the elements using the fastest available sorting algorithm with p processors; **exit.**

Step 2: Partition A_i, for $1 \leq i \leq k$, into segments of size λ each. Here, by a segment we mean a subarray of consecutive locations. (If n_i is not divisible by λ, the last segment may be of a smaller size.) There are $t_i = \lceil \frac{n_i}{\lambda} \rceil$ segments in A_i. So the total number of segments $t = \sum_{i=1}^{k} t_i \leq k + \frac{n}{\lambda} \leq \frac{2n}{\lambda}$. For each segment, let its first element be called its *leader*. Let an array A_i', for $1 \leq i \leq k$, contain the set of leaders from A_i arranged in increasing order.

Step 3: For $1 \leq i, j \leq k$, $i \neq j$, cross-rank the leader arrays A_i' and A_j'. Each leader x now has k ranks, $r_i(x)$ for $1 \leq i \leq k$, assigned to it, $r_i(x)$ being the number of elements less than or equal to x in A_i'.

Step 4: For each leader x, in parallel, add together all $r_i(x)$'s. Consequently, each leader knows the total number of smaller leaders over all arrays; that is, now, all leaders can be placed in an array L of size t, sorted in increasing order.

Step 5: Each leader x knows the two consecutive leaders of A_i, for $1 \leq i \leq k$, that straddle x; note that when $x \in A_i$, the smaller of the two leaders is x itself. In other words, x knows the segment $s_i(x)$ of A_i to which it belongs. For each $x \in L$ and for each array A_i, search $s_i(x)$ in parallel to find the two consecutive elements in it that straddle x. Let $R_i(x)$ be the index in A_i of the smaller one of these two. Thus, $R_i(x)$ is the number of elements in A_i smaller than or equal to x.

Step 6: For each leader x, compute $I(x) = \sum_{i=1}^{k} R_i(x)$, which clearly is the index of x in B, the output array.

REMARK: For any two consecutive elements x and y of L, there can be at most λ elements in each A_i, $1 \leq i \leq k$, that x and y straddle. That is, $R_i(y) - R_i(x) \leq \lambda$, for all i. Hence, $I(y) - I(x) \leq k\lambda$.

Step 7: For every pair of consecutive elements x and y of L, let

$$\delta_i(x) = \begin{cases} R_i(y) - R_i(x) & \text{when } y \notin A_i \\ R_i(y) - R_i(x) - 1 & \text{when } y \in A_i \end{cases}$$

Note that $\delta_i(x)$ is the number of elements that are larger than x but smaller than y in A_i. Let $\langle \Delta_1(x), \ldots, \Delta_k(x) \rangle$ be the prefix sums computed over $\langle \delta_1(x), \ldots, \delta_k(x) \rangle$; $\Delta_0(x)$ is set to zero.

Step 8: Copy $A_i[R_i(x)+1 \ldots R_i(x)+\delta_i(x)]$ to $S_x[\Delta_{i-1}(x)+1 \ldots \Delta_i(x)]$. That is, we isolate all elements that are larger than x but smaller than y into an array S_x.

Step 9: Sort the elements of S_x for each $x \in L$, in parallel, using p processors overall, and place the output in consecutive locations of the subarray of B that begins at index $I(x)+1$.

REMARK: Note that $N(x) = \mid S(x) \mid = I(y) - I(x) - 1 < k\lambda$.

2.2 Preliminaries

The time needed to solve instances of size n, respectively of, prefix summation of b bit numbers, merging, searching, sorting and k-way merging, using p processors in each case, will be denoted by PREFIX(n,p,b), MERGE(n,p), SEARCH(n,p), SORT(n,p) and k_WAY_MERGE(n,p).

We use in the sequel, the following upper bounds that are well known in literature.

Upper Bound 2.1 *[55] When comparison is the only operation allowed on the keys, on a CREW PRAM,* MERGE$(n,p) = O(\log\log n + \frac{n}{p})$.

Upper Bound 2.2 *[55] When comparison is the only operation allowed on the keys, with $p > n$, on a CREW PRAM,* MERGE$(n,p) = O(\log \frac{\log n}{\log(p/n)})$.

Upper Bound 2.3 *[78] When the keys are integers drawn from the range $[0 \ldots m-1]$, on a CREW PRAM,* MERGE$(n,p) = O(\log\log\log m + \frac{n}{p})$.

Upper Bound 2.4 *[78] When the keys are integers drawn from the range $[0 \ldots n-1]$, on a CRCW PRAM,* MERGE$(n,p) = O(\alpha(n) + \frac{n}{p})$.

Upper Bound 2.5 *[46] On an EREW PRAM,* MERGE$(n,p) = O(\log n + \frac{n}{p})$.

Upper Bound 2.6 *[22, 91, 41] On a CRCW PRAM,*

$$\text{PREFIX}(n,p,b) = O\left(\min\left\{\log\frac{b}{\log p}, \log n\right\} + \frac{\log n}{\log\log p} + \frac{n}{p}\right).$$

Upper Bound 2.7 *[55] On an EREW PRAM,* $\textsc{Prefix}(n,p,b) = O(\log n + \frac{n}{p})$.

Upper Bound 2.8 *[55] On a CREW PRAM,* $\textsc{Search}(n,p) = O(\frac{\log n}{\log p})$.

Upper Bound 2.9 *[3, 81, 19] When comparison is the only operation allowed on the keys, on an EREW PRAM,*

$$\textsc{Sort}(n,p) = O\left(\log n + \frac{n \log n}{p}\right).$$

Upper Bound 2.10 *[45] When comparison is the only operation allowed on the keys, on a CRCW PRAM,*

$$\textsc{Sort}(n,p) = O\left(\frac{\log n}{\log(p/n)}(\log\log(p/n))^5 2^{O(\log^* n - \log^*(p/n)+1)} + \frac{\log n}{\log\log p}\right).$$

Upper Bound 2.11 *[7] When the keys can be treated as integers, on an ARBITRARY CRCW PRAM,* $\textsc{Sort}(n,p) = O(\log n + \frac{n \log\log n}{p})$.

Upper Bound 2.12 *[12] When the keys are integers drawn from the range* $[0 \ldots m-1]$, *on an ARBITRARY CRCW PRAM,* $\textsc{Sort}(n,p) = O(\frac{\log n}{\log\log n} + \log\log m + \frac{n \log\log m}{p})$.

Upper Bound 2.13 *[55] The minimum of* n *numbers can be found in* $O(1)$ *time, with* $O(n^{1+\epsilon})$ *processors, on a COMMON CRCW PRAM, for any fixed* $\epsilon > 0$.

Following [40], we say, a set of n objects drawn from a linearly ordered set are padded-sorted if they are arranged in an array of size $O(n)$ in non-decreasing order.

Upper Bound 2.14 *[45] A set of* n *integers drawn from the range* $[0 \ldots n-1]$ *can be stably padded-sorted with a constant padding factor in* $O((\log n)^{1/2}(\log\log n)^{3/2})$ *time and* $O(n \log\log n)$ *operations on an ARBITRARY CRCW PRAM.*

We shall encounter several times in the course of this chapter, the following allocation problem $\mathcal{R}$. Given an array $A = \langle n_1, \ldots, n_k \rangle$, where $\sum_{i=1}^{k} n_i = n$, fill an array $P[1 \ldots n]$ so that, for $1 \leq i \leq n$, $P[i] = j$ iff $\sum_{x=0}^{j-1} n_x < i \leq \sum_{x=0}^{j} n_x$, where $n_0 = 0$.

When p processors are available, the problem $\mathcal{R}$ can be solved as follows. Find the prefix sums $\langle \eta_1, \ldots, \eta_k \rangle$ of A. Let $B[i] = \langle i, 1 \rangle$, and $C[j] = \langle \eta_j, 2 \rangle$ for $1 \leq i \leq n$ and $1 \leq j \leq k$. Cross rank B and C. Now, P can be filled as required in the problem: if the rank of $B[i]$ in C is j then $P[i] = j$. Hence we have the lemma below:

Lemma 2.1 *The problem $\mathcal{R}$ can be solved in $O(\text{PREFIX}(k,p,\log n) + \text{MERGE}(n,p))$ time.*

We assume that on all computational models under consideration here, the upper bounds for merging, prefix summing and sorting satisfy the the "regularity" condition that larger problem instances with the same processor-advantage (p/n) will take more time. That is, $\text{MERGE}(n_1, n_1\beta) \leq c\text{MERGE}(n_2, n_2\beta)$, for $n_1 < n_2$ and a constant $c > 0$; similarly for sorting and prefix summing.

2.3 Comparison Based Multiway Merging

2.3.1 The schema on PRAMs with Concurrent Reads

First, we consider some aspects common to CREW and CRCW PRAM implementations of the algorithm schema presented in Section 2.1. Let us set $\lambda = k^2$.

In Step 1, where we are sorting all the elements together, $n < k^3$. Since, solving a larger problem instance with the same processor advantage cannot be easier, the time taken by this step is $O(\text{SORT}(n,p)) = O(\text{SORT}(k^3, \frac{pk^3}{n}))$.

We can both partition the input arrays and form the leader arrays in $O(1)$ time, using n processors, once they are assigned one per element. But here we have p processors. Assign one processor for every $\Theta(\frac{n}{p})$ elements of the input, using Lemma 2.1. Each processor can now sequentially select leaders from the elements assigned to it. So, Step 2 takes $O(\frac{n}{p} + \text{MERGE}(n,p) + \text{PREFIX}(k,p,\log n))$ time.

In Step 3, $\frac{k(k-1)}{2}$ merges, each of size at most t, are to be performed in parallel. Assign $\Theta(\frac{p}{k^2})$ processors per merge instance. Solve each instance independently. The time taken is clearly $O(\text{MERGE}(t, \frac{p}{k^2}))$.

Steps 4 and 6 each requires every leader to perform an addition of size k, of $\log n$ bit numbers. Use Lemma 2.1 to assign $\Theta(\frac{p}{t})$ processors per leader. Find the sums using a prefix summation algorithm in $O(\text{PREFIX}(k, \frac{p}{t}, \log n))$ time. The total time taken is $O(\text{MERGE}(n,p) + \text{PREFIX}(k, \frac{p}{t}, \log n))$.

Each leader has to perform k searches in Step 5, each over a segment of size k^2; that is, a total of tk searches have to be performed in parallel. At most $\lfloor \frac{p}{tk} \rfloor$ processors are available per search. If k processors were to be used per search, the time taken would be $\text{SEARCH}(k^2, k)$, which is $O(\frac{\log k^2}{\log k}) = O(1)$, by Upper Bound 2.8. With $\lfloor \frac{p}{tk} \rfloor$ processors

per search, hence, we can finish the step in $O(\frac{k}{p/tk} + \text{SEARCH}(k^2, k))$ time. As $tk^2 < 2n$ (see Step 2 of the schema), the time needed for Step 5 is $O(\frac{n}{p} + \text{SEARCH}(k^2, k)) = O(\frac{n}{p})$.

Step 7 requires every leader to perform an addition of size k, of $\log k$ bit numbers. With a processor allocation similar to that in Step 4, the sums can be found in $O(\text{PREFIX}(k, \frac{p}{t}, \log k))$ time. Since the leaders are now merged into the array L, the allocation incurs only $O(1)$ time.

In Step 8, we prepare the sorting instances. Each leader x, for $1 \leq i \leq k$, if $\delta_i(x) \neq 0$, then tags element $A_i[R_i(x)+1]$ with the address of the location $S_x[\Delta_{i-1}(x)+1]$; note that $A_i[R_i(x)+1]$ is to be copied to $S_x[\Delta_{i-1}(x)+1]$. Assign $\Theta(\frac{p}{t})$ processors to each segment as in Step 7; and for each element find the largest tagged element smaller than it in its array. As the second element in the segment is guaranteed to be tagged, the first being a leader, this does not require any inter-segment information access; a prefix minima over each segment, done in parallel, will do. Thus, each element can compute the address of the cell it must be copied to. Finally, the copying itself can be done in $O(\frac{n}{p})$ time. So, the total time taken is $O(\frac{n}{p} + \text{PREFIX}(k^2, \frac{p}{t}, 2\log k))$.

In Step 9, we again assign $\Theta(\frac{p}{t})$ processors per segment, as in Step 7. For each leader x, dedicate to x, a fraction $\Theta(\frac{\delta_i(x)}{k^2})$ of the processors assigned to $s_i(x)$, the segment of A_i to which x belongs, by carrying out another prefix summation over each segment of A_i, for $1 \leq i \leq k$. That is each leader x is now assigned $\Theta(\frac{N(x)p}{n})$ processors. Since the prefix summation can be clubbed with that of Step 8, this step can be finished in $O(\text{SORT}(k^3, \frac{pk^3}{n}))$ time.

On a PRAM that allows concurrent reads, $\text{SORT}(m, q)$, $\text{PREFIX}(m, q, b)$ and $\text{MERGE}(m, q)$ are all at most $O(\frac{m}{q} + \log m)$. Hence, for a constant c,

$$\text{SORT}(m^{c+1}, qm^c) = \Theta(\text{SORT}(m, q))$$

$$\text{PREFIX}(m^{c+1}, qm^c, b) = \Theta(\text{PREFIX}(m, q, b))$$

and

$$\text{MERGE}(m^{c+1}, qm^c) = \Theta(\text{MERGE}(m, q)).$$

Thus, we have the following lemma:

Lemma 2.2 *When concurrent reads are allowed, k sorted arrays of a total size n, can be merged in*

$$O\left(\text{SORT}(k, \tfrac{pk}{n}) + \text{PREFIX}(k, \tfrac{p}{kt}, \log n) + \text{MERGE}(n, p) + \frac{n}{p}\right)$$

time using p processors.

On a CREW PRAM, we can sort n elements in $\text{SORT}(n,p) = O(\frac{n \log n}{p} + \log n)$ time (Upper Bound 2.9), find the prefix sum of n b-bit numbers in $\text{PREFIX}(n,p,b) = O(\frac{n}{p} + \log n)$ time (Upper Bound 2.7), and merge two arrays of a total size n in $\text{MERGE}(n,p) = O(\frac{n}{p} + \log\log n)$ time (Upper Bound 2.1).
Thus, we have, when $p \leq n$,

$$\text{SORT}(k, \tfrac{pk}{n}) = O\left(\frac{k \log k}{pk/n} + \log k\right) = O\left(\frac{n \log k}{p} + \log k\right) \tag{2.1}$$

$$\text{PREFIX}(k, \tfrac{p}{kt}, \log n) = O\left(\frac{k^2 t}{p} + \log k\right) = O\left(\frac{n}{p} + \log k\right) \tag{2.2}$$

Applying Lemma 2.2, we have the following theorem:

Theorem 2.1 *On a PRAM that allows concurrent reads, using p processors, k sorted arrays of a total size n can be merged in time*

$$k\text{\_WAY\_MERGE}(n,p) = O\left(\log k + \log\log n + \frac{n \log k}{p}\right)$$

Theorem 2.2 *On a CREW PRAM, using $p > n$ processors, k sorted arrays of a total size n can be merged in time*

$$k\text{\_WAY\_MERGE}(n,p) = O\left(\log\left(\frac{\log n}{\log(p/n)}\right) + \log k\right)$$

Proof: Use Upper Bound 2.9 for sorting, Upper Bound 2.2 for merging, and Upper Bound 2.7 for prefix summation, in Lemma 2.2. □

Corollary 2.1 *On a CREW PRAM,*

$$k\text{\_WAY\_MERGE}(n,p) = O(\text{SORT}(k, \tfrac{pk}{n}) + \text{MERGE}(n,p)).$$

Theorem 2.3 *On a CRCW PRAM, k sorted arrays of a total size n can be merged in*

$$O\left(\log\left(\frac{\log n}{\log(p/n)}\right) + \text{SORT}(k, \tfrac{pk}{n})\right)$$

time, using $p > n$ processors.

Proof: On a CRCW PRAM, with $p > n$ processors, we can find the prefix sum of n b-bit numbers in

$$\text{PREFIX}(n,p,b) = O\left(\min\left\{\log\frac{b}{\log p}, \log n\right\} + \frac{\log n}{\log\log p} + \frac{n}{p}\right)$$

time (Upper Bound 2.6) and merge two arrays of a total size n in $\text{MERGE}(n,p) = O(\log\frac{\log n}{\log(p/n)})$ time (Upper Bound 2.2). If, in addition, we use the best known CRCW PRAM sorting algorithm, by Lemma 2.2, the algorithm schema can be implemented in time

$$O\left(\log\frac{\log n}{\log(p/n)} + \min\left\{\log\frac{\log n}{\log(p/tk)}, \log k\right\} + \frac{\log k}{\log\log(p/tk)} + \text{SORT}(k, \tfrac{pk}{n})\right)$$

$$= O\left(\log\frac{\log n}{\log(p/n)} + \frac{\log k}{\log\log(pk/n)} + \text{SORT}(k, \tfrac{pk}{n})\right).$$

Since $\text{SORT}(k, \frac{pk}{n}) = \Omega(\frac{\log k}{\log\log(pk/n)})$, the theorem follows. □

Corollary 2.2 *On a CRCW PRAM, k sorted arrays of a total size n, can be merged in* k-WAY_MERGE(n,p)

$$= O\left(\log\left(\frac{\log n}{\log(p/n)}\right) + \frac{\log k}{\log(p/n)}(\log\log(p/n))^5 2^{O(\log^* k - \log^*(p/n)+1)} + \frac{\log k}{\log\log(pk/n)}\right)$$

time, using $p > n$ processors,

Proof: Apply Upper Bound 2.10. □

Corollary 2.3 *On a CRCW PRAM,*

$$k\text{-WAY\_MERGE}(n,p) = O(\text{SORT}(k, \tfrac{pk}{n}) + \text{MERGE}(n,p)).$$

2.3.2 The Schema on an EREW PRAM

Theorem 2.4 *On an EREW PRAM, with p processors, k sorted arrays of a total size n can be merged in time*

$$k\text{-WAY\_MERGE}(n,p) = O\left(\log n + \frac{n\log k}{p}\right)$$

Proof: We consider an implementation of the algorithm schema of Section 2.1. Again, we take $\lambda = k^2$. Each step of the schema can be implemented on an EREW PRAM with $q = \lfloor\frac{n\log k}{\log n}\rfloor$ processors, as follows:

Step 1: Since in this step, $k > n^{1/3}$, $\log k = \Theta(\log n)$ and hence, Upper Bound 2.9 can be used to sort the input elements $O(\frac{n \log n}{q} + \log n) = O(\log n)$ time.

Step 2: We can both partition the input arrays and form the leader arrays in $O(1)$ time, using n processors once they are assigned one per element. We allocate the processors uniformly over the input using Lemma 2.1. This would take $O(\log n + \frac{n}{q})$ time. Each processor can sequentially select leaders from the elements assigned to it. So, the total time taken for this step is $O(\log n + \frac{n}{q}) = O(\log n)$.

Step 3: Make k copies of each leader array; that is, a total of $kt \leq \frac{n+k^3}{k} \leq \frac{2n}{k}$ entries have to be made. With $\Theta(\frac{k}{\log k})$ processors per leader, the copies can be made in $O(\log k)$ time. Distribute $\lfloor \frac{k}{\log n} \rfloor$ processors per leader as in Step 2 and finish copying in $O(\log n)$ time. The total number of processors used is at most $\frac{tk}{\log n} \leq \frac{2n}{k \log n} \leq \frac{n}{\log n} \leq q$. For merging a pair of leader arrays A'_i and A'_j, for $1 \leq i, j \leq k$, $i \neq j$, use the j-th copy of A'_i and the i-th copy of A'_j. Since, on an EREW PRAM with p processors two arrays of a total size n can be merged in $O(\log n + \frac{n}{p})$ time (see Upper Bound 2.5), here with $\lfloor \frac{t}{\log n} \rfloor$ processors per pair, the time is $O(\log n)$. Since, there are $\frac{k(k-1)}{2}$ merge instances, the total number of processors used is at most $\frac{tk^2}{2 \log n} \leq \frac{n}{\log n} \leq q$. Note that the processor allocation incurs only $O(1)$ time in the case of merging. Thus, the total time required for this step is $O(\log n)$.

Step 4: Once the k ranks for each leader have been found, add the cross-ranks together in $O(\log n)$ time with $\lfloor \frac{k}{\log n} \rfloor$ processors per leader (see Upper Bound 2.7).

Step 5: Positioning the leaders in their corresponding segments has the difficulty that several leaders may want to search the same segment simultaneously. To avoid concurrent reads, make k copies of L, the merged array of all leaders. Let these copies be, L_i for $1 \leq i \leq k$. With $\lfloor \frac{k}{\log n} \rfloor$ processors per element of L copying can be finished in $O(\log n)$ time. Merge L_i with A_i. Assign $\lfloor \frac{t+n_i}{\log n} \rfloor$ processors for the i-th merge. Note that this processor distribution may require $O(\log n)$ time, as in Step 2. The merges and hence the cross-ranks can be found in $O(\log n)$ time (see Upper Bound 2.5). The total number of operations used is $\sum_{i=1}^{k} O(t + n_i) = O(tk + n) = O(n)$.

Step 6,7: Now that for all $x \in L$, all $R_i(x)$'s have been found, computing $I(x)$ involves prefix sums computation, and takes at most $O(\log n)$ time using Upper Bound 2.7.

Computation of $\Delta_i(x)$'s for all $l \in L$ also requires prefix sums computation, and similarly, can be done in $O(\log n)$ time.

Step 8: We prepare the sorting instances in this step. Each leader x, for $1 \leq i \leq k$, knows $\delta_i(x)$, the number of elements to be copied from A_i as well as the base address $S_x[\Delta_{i-1}(x)+1]$, where the smallest of those elements must go. If we have one processor per element, that is, n processors overall, each element can be informed the address of the location to which it must be copied in $O(\log k)$ time using a broadcast over a distance of at most $\delta_i(x) \leq k^2$ and subsequently, the copying itself can be done in $O(1)$ time. With q processors this step would hence take $O(\frac{n \log k}{q}) = O(\log n)$ time.

Step 9: As n elements can be sorted on an EREW PRAM in $O(\log n + \frac{n \log n}{q})$ time (see Upper Bound 2.9), with $\lfloor \frac{N(l)q}{n} \rfloor$ processors assigned to x, we can sort $S(x)$ for each $x \in L$ in parallel in $O(\frac{N(x) \log N(x)}{q}) = O(\log n)$ time. The processor allocation here would require a prefix summation over L that takes $O(\log n)$ time.

That is, with $q = \lfloor \frac{n \log k}{\log n} \rfloor$ processors, k-way merging can be done in $O(\log n)$ time on an EREW PRAM. Hence the theorem. □

Corollary 2.4 *On an EREW PRAM,*

$$k\text{-WAY\_MERGE}(n,p) = O(\text{SORT}(k, \tfrac{pk}{n}) + \text{MERGE}(n,p)).$$

From Corollaries 2.1, 2.3, 2.4,

Corollary 2.5 *On a PRAM,*

$$k\text{-WAY\_MERGE}(n,p) = O(\text{SORT}(k, \tfrac{pk}{n}) + \text{MERGE}(n,p)).$$

2.3.3 Lower Bounds

In this section we prove lower bounds on time for comparison-based k-way merging. Our main results are Theorem 2.5 for EREW PRAM, Theorem 2.6 for CREW PRAM, and Theorem 2.7 for CRCW PRAM.

Lemma 2.3 *On all models at least as powerful as EREW PRAM,*

$$k\text{-WAY\_MERGE}(n,p) = \Omega(\text{MERGE}(n,p)).$$

Proof: Suppose that A and B are two sorted arrays of size $n/2$ each. Divide both A and B into $k/2$ segments of a non-zero size each. Here, by a segment we mean a subarray of consecutive locations; each of the segments hence is a sorted array. These k segments together form an instance of k-way merging of size n, a solution to which is clearly equivalent to the merge of A and B. Hence the lemma. □

Lemma 2.4 *On all models at least as powerful as EREW PRAM,*

$$\text{MERGE}(n,p) = \Omega(\text{SEARCH}(n,p)).$$

Proof: Given an instance of search, $A[1 \ldots n]$ with search key x, and p processors, for two arrays B and C of size $\frac{n}{2}+1$ each, for $i = 1, \ldots, \frac{n}{2}$, let $B[i] = A[i]$ and $C[i+1] = A[\frac{n}{2}+i]$. If $x \geq A[\frac{n}{2}]$ then let $B[\frac{n}{2}+1] = x, C[1] = -\infty$, else if $x < A[\frac{n}{2}]$ then let $B[\frac{n}{2}+1] = +\infty, C[1] = x$. Here $-\infty$ and $+\infty$ are distinct keys respectively smaller and larger than all other keys. Both of B and C being sorted, merging them will answer the search. Thus, $\text{MERGE}(n+2,p) = \Omega(\text{SEARCH}(n,p))$, and hence the lemma. □

Corollary 2.6 *On all models at least as powerful as EREW PRAM,*

$$k\text{\_WAY\_MERGE}(n,p) = \Omega(\text{SEARCH}(n,p)).$$

Theorem 2.5 *On an EREW PRAM, the time required for comparison-based merging of k sorted arrays of a total size n, using $p \geq n$ processors, is*

$$k\text{\_WAY\_MERGE}(n,p) = \Omega(\log n).$$

Proof: Snir [93] has shown that $\text{SEARCH}(n,p) = \Omega(\log n)$, on an EREW PRAM for any value of p. Thus the theorem follows from Corollary 2.6. □

Lemma 2.5 *On a CREW PRAM, even when the keys are single bit numbers,*

$$k\text{\_WAY\_MERGE}(n,p) = \Omega(\log k).$$

Proof: Given an instance $(x_1, \ldots, x_k)$ of OR of size k, construct sorted arrays Z_i, for $1 \leq i \leq k$, of size n/k each as follows: set $Z_i[\frac{n}{k}]$ to x_i and all of $Z_i[j]$ for $1 \leq j < \frac{n}{k}$, to zero. Find the merge of all Z_i's. If the last number of the output is one, then the OR is one, else it is zero. Cook et.al. [23] have shown that $\Omega(\log k)$ is a lower bound for finding the OR of k-bits on a CREW PRAM. □

Theorem 2.6 *On a CREW PRAM, the time required for comparison-based merging of k sorted arrays of a total size n, using $p \geq n$ processors, is*

$$k\text{-WAY\_MERGE}(n,p) = \Omega\left(\log\left(\frac{\log n}{\log(p/n)}\right) + \log k\right).$$

Proof: Merging two sorted arrays of size $n/2$ each with $p \geq n$ processors requires at least

$$\Omega\left(\log \frac{\log n}{\log(p/n)}\right)$$

time on the parallel comparison model [55, 14], and hence on a CREW PRAM, because the latter is strictly weaker than the former. Hence, by Lemma 2.3,

$$k\text{-WAY\_MERGE}(n,p) = \Omega\left(\log \frac{\log n}{\log(p/n)}\right).$$

For the second term, we use Lemma 2.5. □

Lemma 2.6 *The number of comparisons required for merging k sorted arrays of a total size n, in t comparison rounds on the parallel comparison model is $\Omega(t(nk^{1/t} - n))$.*

Proof: Consider the problem $\mathcal{P}$ of independently sorting $\lfloor n/k \rfloor$ arrays of k elements each. Let $\langle x_1^i, \ldots, x_k^i \rangle$ be the i-th sorting instance, for $1 \leq i < \lfloor n/k \rfloor$. Since, an adversary can always make sure that each element of the i-th instance is smaller than every element of the $(i+1)$-th instance, the minimum number of comparisons required for solving $\mathcal{P}$ is asymptotically the same as $\lfloor n/k \rfloor$ times the minimum number of comparisons required for sorting one instance. Azar and Vishkin [9] have shown that the minimum number of comparisons required to sort k elements in t comparison rounds is $\Omega(t(k^{1+1/t} - k))$.

Now, construct a 2-dimensional array S of size $\lfloor n/k \rfloor \times k$ and let $S[i,j] = \langle i, x_j^i \rangle$. Comparing its elements lexicographically, S has the property that each element of the i-th row is smaller than every element of the $(i+1)$-th row. Hence, $\mathcal{P}$ can be solved by k-way merging all columns of S. The number of comparisons between the composite keys of S, that such an algorithm uses, should be at least as much as the minimum number of comparisons between the original keys that are required to solve $\mathcal{P}$. Hence the lemma. □

Lemma 2.7 *The number of processors required for merging k sorted arrays of a total size n, in t units of time on the CRCW PRAM model is*

$$\Omega(\lfloor \frac{n}{k} \rfloor 2^{ck^{1/t}}).$$

Proof: Consider the problem $\mathcal{P}$, dealt in the proof of the above lemma, again. The minimum number of processors required for solving $\mathcal{P}$ is asymptotically the same as $\lfloor n/k \rfloor$ times the minimum number of processors required for sorting one instance. Beam and Hastad [10] show that sorting k keys in $O(t)$ time on a CRCW PRAM requires $\Omega(2^{ck^{1/t}})$ processors, for a constant c. □

Theorem 2.7 *On a CRCW PRAM, the time required for comparison-based merging of k sorted arrays of a total size n, using $p \geq n$ processors, is*

$$k\text{\_WAY\_MERGE}(n,p) = \Omega\left(\log\left(\frac{\log n}{\log(p/n)}\right) + \frac{\log k}{\log(1+p/n)} + \frac{\log k}{\log\log(pk/n)}\right).$$

Proof: The first term can be proved as in Theorem 2.6. Equating k_WAY_MERGE(n,p) with t, the second and third terms follow from Lemma 2.6 and Lemma 2.7 respectively. □

Corollary 2.7 *On a PRAM,*

$$k\text{\_WAY\_MERGE}(n,p) = \Omega(\text{SORT}(k, \tfrac{pk}{n}) + \text{MERGE}(n,p)).$$

Proof: On a PRAM that does not allow concurrent writes, SORT$(k, \frac{pk}{n}) = \Theta(\log k)$. Hence, from Lemma 2.4, Theorem 2.5 and Theorem 2.6, the corollary can be shown to hold for EREW and CREW PRAMs. From Theorem 2.7, the CRCW PRAM model also satisfies the result. □

From Corollaries 2.5 and 2.7, we have the following theorem:

Theorem 2.8 *On a PRAM,*

$$k\text{\_WAY\_MERGE}(n,p) = \Theta(\text{SORT}(k, \tfrac{pk}{n}) + \text{MERGE}(n,p)).$$

2.4 Multiway Merging of Integers

2.4.1 An Upper Bound from the Schema

On an ARBITRARY CRCW PRAM, we can sort n integers in SORT$(n,p) = O(\log n + \frac{n \log\log n}{p})$ time (Upper Bound 2.11). Thus,

$$\text{SORT}(k, \tfrac{pk}{n}) = O\left(\frac{k\log\log k}{pk/n} + \log k\right) = O\left(\frac{n\log\log k}{p} + \log k\right) \qquad (2.3)$$

Using this, together with, Eq. (2.2) and Upper Bound 2.1, we get:

Lemma 2.8 *On an ARBITRARY CRCW PRAM, when the array elements are given to be integers, using p processors, k sorted arrays of a total size n can be merged in time*

$$k\text{-WAY-MERGE}(n,p) = O\left(\log k + \log\log n + \frac{n\log\log k}{p}\right)$$

However, if the integers are from the range $[0\ldots m-1]$, we may use the algorithm of Upper Bound 2.12 to sort integers in $\text{SORT}(n,p) = O(\frac{\log n}{\log\log n} + \log\log m + \frac{n\log\log m}{p})$ time. Thus,

$$\begin{aligned}\text{SORT}(k,\tfrac{pk}{n}) &= O\left(\frac{k\log\log m}{pk/n} + \frac{\log k}{\log\log k} + \log\log m\right)\\ &= O\left(\frac{n\log\log m}{p} + \frac{\log k}{\log\log k} + \log\log m\right),\end{aligned}$$

For prefix summation, we can use the faster algorithm of Upper Bound 2.6. Hence,

$$\begin{aligned}\text{PREFIX}(k,\tfrac{p}{kt},\log n) &= O\left(\min\left\{\log\frac{\log n}{\log(p/tk)},\log k\right\} + \frac{\log k}{\log\log(p/tk)} + \frac{k}{p/tk}\right)\\ &= O\left(\log\log n + \frac{\log k}{\log\log k} + \frac{n}{p}\right)\end{aligned}$$

Combining these with Upper Bound 2.1, from Lemma 2.2, we have the following lemma:

Lemma 2.9 *On an ARBITRARY CRCW PRAM, when the array elements are integers drawn from the range $[0\ldots m-1]$, using p processors, k sorted arrays of a total size n can be merged in time*

$$O\left(\frac{\log k}{\log\log k} + \log\log mn + \frac{n\log\log m}{p}\right)$$

Alternatively, we can use the $\text{MERGE}(n,p) = O(\log\log\log m + \frac{n}{p})$ time algorithm for merging integers in the range $[0\ldots m-1]$ (Upper Bound 2.3). Together, with Eq. (2.3) and Eq. (2.2), this gives, from Lemma 2.2:

Lemma 2.10 *On an ARBITRARY CRCW PRAM, when the array elements are assumed to be integers drawn from the range $[0\ldots m-1]$, using p processors, k sorted integer arrays of a total size n can be merged in time*

$$O\left(\log k + \log\log\log m + \frac{n\log\log k}{p}\right)$$

Combining Lemmas 2.8, 2.9, and 2.10, we have:

Theorem 2.9 *On an ARBITRARY CRCW PRAM, when the array elements are assumed to be integers drawn from the range* $[0 \ldots m-1]$, *using* p *processors,* k *sorted integer arrays of a total size* n *can be merged in time* k-WAY-MERGE$(n,p) =$

$$O\left(\frac{\log k}{\log\log k} + \log\log(\min\{n, \log m\}) + \min\left\{\log k + \frac{n\log\log k}{p}, \log\log mn + \frac{n\log\log m}{p}\right\}\right)$$

2.4.2 Lower Bounds

Lemma 2.11 *On a CRCW PRAM, even when the keys are single bit numbers, for* $p \geq n$,

$$k\text{-WAY-MERGE}(n,p) = \Omega\left(\frac{\log k}{\log\log p}\right).$$

Proof: Given an instance $(x_1, \ldots, x_k)$ of Parity of size k, construct sorted arrays Z_i, for $1 \leq i \leq k$, of size n/k each as follows: set $Z_i[\frac{n}{k}]$ to x_i and all of $Z_i[j]$ for $1 \leq j < \frac{n}{k}$, to zero. Find the merge of all Z_i's. If the last zero of the output is at index m, then the number of ones in this input is $n - m$. Since the lower bound for finding the parity of k bits with p processors on a CRCW PRAM is $\Omega(\frac{\log k}{\log\log p})$ [10], the lemma follows. □

Given an instance of k-way merging of n single bits numbers, the solution can be obtained by adding together the number of zeroes in each array. This takes PREFIX$(k,p,\log n)$ time, which, by Upper Bound 2.6, is $O(\frac{\log k}{\log\log p})$. That is, the lower bound of the above lemma is tight when the integers are of single bits.

Theorem 2.10 *On a CRCW PRAM, with* $p \geq n$ *processors, the time required to merge* k *sorted arrays of integers from the range* $[0 \ldots \frac{n}{k} - 1]$ *of a total size* n *is* $\Omega(\frac{\log k}{\log\log(pk/n)})$.

Proof: Similar to the problem $\mathcal{P}$ dealt in the proof of Lemma 2.6, consider a problem $\mathcal{Q}$ of independently sorting arrays $(x_1^i, \ldots, x_k^i)$, of single bit numbers, for $1 \leq i < \lfloor n/k \rfloor$. Sorting each row independently, with p processors overall, will require $\Omega(\frac{\log k}{\log\log(pk/n)})$ time (see Lemma 2.7).

Replace each x_j^i by $y_j^i = 2i + x_j^i$. Thus y_j^i are from the range $[0 \ldots \frac{2n}{k} - 1]$. Construct a two dimensional array S with $\lfloor n/k \rfloor$ rows and k columns and let $S[i,j] = y_j^i$. Clearly, S has the property that each element of the i-th row is smaller than every element of the $(i+1)$-th row. Since each column of S is a sorted array, $\mathcal{Q}$ can be solved by k-way merging all columns of S.

Hence the theorem. □

2.4.3 Two More Upper Bounds

Integers in the range $[0 \ldots m-1]$ can be k-way merged as follows.

Let $Less_j[i]$ be the the total number of elements that are smaller than integer i in A_j, the j-th input array, for $0 \leq i < m$ and $1 \leq j \leq k$; $Less_j[i]$ can be computed by merging the array $\langle 0, \ldots, m-1 \rangle$ with A_j. Compute $S_i = \sum_{j=1}^{k} Less_j[i]$, for all i, in parallel. Let $Count_j[i]$ be the the number of occurrences of integer i in A_j; if $First_j[i]$ and $Last_j[i]$ are the indices of, respectively, the first and last occurrences of i in A_j, then $Count_j[i] = Last_j[i] - First_j[i] + 1$. For each integer i, compute $PS\_Count[i,j] = \sum_{l=1}^{j} Count_l[i]$. Now, copy the h-th occurrence of i in A_j to the location numbered $S_i + PS\_Count[i, j-1] + h$ of the output.

With p processors, the merging in the computation of *Less* can be done in $O(\alpha(m) + \frac{m}{p})$ time, by Upper bound 2.4. For obtaining *First* and *Last*, each input key has only to compare itself with the adjacent keys in its array; that is, with n processors, *Count* can be computed in $O(1)$ time. With p processors, hence, the computation of *Count*, as well as the final copying can be done in $O(\frac{n+mk}{p})$ time. That is, the total time taken is $O(\frac{n+mk}{p} + \alpha(m) + \textsc{Prefix}(k, \frac{p}{m}, \log n))$. Using, Upper Bound 2.6 for prefix summation, hence, we have the following theorem.

Theorem 2.11 *On a CRCW PRAM, with p processors, k sorted arrays of integers from the range $[0 \ldots \frac{n}{k} - 1]$, of a total size n, can be merged in*

$$k\text{-}\textsc{Way\_Merge}(n,p) = O\left(\frac{n}{p} + \alpha(n/k) + \frac{\log k}{\log\log(pk/n)} + \log\frac{\log n}{\log(pk/n)}\right)$$

time.

When the output does not have to be given in an array, we can do better, as the following theorem indicates.

Theorem 2.12 *Given k sorted arrays of integers from the range $[0 \ldots n-1]$, an ordered chain of the integers can be obtained in $O(\frac{\log k}{\log\log k} + \alpha(n))$ time with $\lfloor \frac{n \log\log k}{\alpha(n) + \log k/\log\log k} \rfloor$ processors on an ARBITRARY CRCW PRAM.*

Proof: Let $p = \lfloor \frac{n \log\log k}{\alpha(n) + \log k/\log\log k} \rfloor$. Follow the algorithm schema of Section 2.1, with the following minor variations. Choose $\lambda = k^2$.

Step 4: Use the cross ranks computed in Step 3 to find for each leader x, its largest smaller x_i in A'_i; that is, $x_i = A'_i[r_i(x)]$ when $x \notin A'_i$ and $x_i = A'_i[r_i(x)-1]$ when $x \in A'_i$. Find the largest among all x_i's. We now have an ordered chain of all leaders.

Merging two arrays once they are cross ranked takes only $O(1)$ time with linear number of processors even on an EREW PRAM [55]. With k^2 processors assigned to x, the largest among all x_i's also can be found in $O(1)$ time (see Upper Bound 2.13). Hence, with $\lfloor \frac{p}{t} \rfloor$ processors per leader this step takes at most $O(\frac{n}{p})$ time.

Step 6: Do not perform the addition.

Step 9: For each leader x, padded-sort S_x into Z_x. Ordered-chain Z_x; this coupled with the ordered chain of all leaders, obtained in Step 4, gives an ordered-chaining of all input elements.

By Upper Bound 2.14, padded-sorting can be done in $O(\frac{\log k}{\log\log k})$ time with $O(n \log\log k)$ operations overall. Ordered chaining Z_x can be done in $O(\log^* k)$ time [79, 83] with $O(n)$ operations overall, by replacing every non-zero element by its index in Z_x. Coupling the local ordered chains with the global one can be done in $O(1)$ time with $O(n)$ operations.

Hence, the total time taken is,

$$O\left(\frac{\log k}{\log\log k} + \textsc{Prefix}(k, \tfrac{p}{tk}, \log k) + \textsc{Prefix}(k, p, \log n) + \textsc{Merge}(t, \tfrac{2p}{k^2}) + \frac{n \log\log k}{p}\right).$$

Using Upper Bound 2.4 for merging and Upper Bound 2.6 for prefix summation, we have the theorem. □

Chapter 3

Parallel Merge Sort

Given an array of n elements to sort, we can proceed as follows: divide the array into δ segments of equal size, sort each segment recursively, and δ-way merge the sorted segments. Following this framework, and using the multiway merging schema of Chapter 2, a parallel merge sorting algorithm is obtained. This algorithm runs in $\frac{\log n}{\log \alpha} \cdot 2^{O(\log^*(\log n/\log \alpha))}$ steps on the parallel comparison model.

Valiant, who introduced the parallel comparison model, also gave, on that model, a simple $O(\log(\frac{\log n}{\log \alpha}))$ time algorithm for merging two arrays of a total size n, with $n\alpha$ processors, $2 \leq \alpha \leq n$ [96]. Borodin and Hopcroft [14] proved a matching lower bound. For sorting n elements with $n\alpha$ processors on the parallel comparison model, Azar and Vishkin proved a lower bound of $\Omega(\frac{\log n}{\log \alpha})$ [9]. But, unlike merging, no simple sorting algorithm matching the lower bound, for all values of α, has been found. Alon indeed obtained an algorithm [9] that matches the lower bound, but being an adaptation of the AKS sorting network [3, 81, 82] of Ajtai, Komlós, and Szemerédi, this algorithm is not simple and has large constant factors. Subsequently, Cole [19] gave a simple merge sort algorithm with small constant factors; this algorithm runs in $O(\frac{\log n}{\log \alpha} \cdot \log\log \alpha)$ steps on the parallel comparison model. Note that except for very small values of α, our algorithm is closer to the lower bound than Cole's merge sort; moreover it is very different from Cole's merge sort. In our algorithm too, vis-a-vis Alon's algorithm, the constant factors involved are small.

On a CREW PRAM, Cole's merge sort is optimal—with n processors it can be implemented in $O(\log n)$ time; on a CREW PRAM, with any number of processors, $\Omega(\log n)$ is a lower bound on sorting [23]. In contrast, the proposed algorithm, *when*

implemented on a CREW PRAM with n processors, runs in $\log n \cdot 2^{O(\log^* n)}$ time; this implementation may be thought simpler than Cole's merge sort because it does not use an intricate pipelining scheme. On a Concurrent Read Concurrent Write (CRCW) PRAM This algorithm runs in $\frac{\log n}{\log \alpha} \cdot 2^{O(\log^*(\log n/\log \alpha))}$ steps on the parallel comparison model.

3.1 A Procedure

In this section we describe a procedure SEGREGATE. This procedure is the same as our k-way merging schema, except that the last step of the schema is not included here. For the implementation details not covered here, please refer to Chapter 2.

In a procedure call "SEGREGATE(A, m, δ)", A is an array of m elements with δ sorted subarrays A_i, $1 \leq i \leq \delta$, of size m/δ each, given one after another contiguously. The output of the call is a permutation of the input elements, given in the array A itself, that would have m/δ (not necessarily sorted) subarrays $G_j(A)$ of size $g_j(A)$, $g_j(A) \leq \delta^2$, $1 \leq j \leq m/\delta$, given one after another contiguously, so that all elements of a subarray are larger than every element of a preceding subarray.

The procedure is described below. The number of processors available is $m\beta$, $\beta > 0$.

Procedure SEGREGATE(A, m, δ)

For $1 \leq i \leq \delta$, let every δ-th element from A_i be called a leader; the leaders from A_i, while given in the same order as they appear in A_i, form, let us say, a leader array A'_i. We call the set of elements in A_i between any two consecutive of its leaders, a segment. There are m/δ^2 leaders in each A_i. In parallel, merge every pair of leader arrays by assigning $m\beta/\delta^2$ processors per pair. Now, for $1 \leq i, j \leq \delta$, each element of A'_i knows its rank in A'_j. With $\beta\delta$ processors per leader, in parallel for each leader x, add together the ranks of x; all the leaders can now be arranged in sorted order, say, in an array, L.

Each leader x now knows the segment $s_i(x)$ of A_i to which it belongs. For each $x \in L$ and for each array A_i, search $s_i(x)$ in parallel to find the two consecutive elements in it that straddle x; assign β processors per leader per search. Thus, each $x \in L$ has now found its rank $R_i(x)$ in every A_i. For each $x \in L$, find the prefix sums over $R_i(x)$, $1 \leq i \leq \delta$; x now knows its index in the output array. Also, for

every consecutive $x, y \in L$, we can now tag the first element (where one exists) from the interval (x, y) in A_i with its index in the output array. Now, a prefix minima computation over each segment is enough to inform each element of A of its index in the output array. Permute the elements in A according to the indices found.

The following lemma is now apparent:

Lemma 3.1 *The procedure call "*SEGREGATE(A, m, δ)*" takes*

$$O(\text{PREFIX}(\delta, \delta^2\beta, \log m) + \text{MERGE}(m, m\beta) + \text{SEARCH}(\delta, \beta))$$

time with $m\beta$ processors.

3.2 The Algorithm on a CREW PRAM

First we present the CREW PRAM implementation of the algorithm. We construct a sequence of sorting algorithms, the $(\log^* n - 2)$-th one of the sequence achieving the claimed resource bounds, using the following idea:

> Given an $O(c_r \log n \log^{(r)} n)$ time sorting algorithm $\mathcal{A}$, $r \geq 2$, for $\frac{c_{r+1}}{c_r} = O(1)$, an $O(c_{r+1} \log n \log^{(r+1)} n)$ time sorting algorithm $\mathcal{B}$ can be obtained.

A high level description of the new algorithm $\mathcal{B}$ would be as follows:

> Divide the input array into subarrays of size δ^2 each, for a parameter δ to be chosen. Sort each subarray using algorithm $\mathcal{A}$. Now it remains only to merge the $\frac{n}{\delta^2}$ sorted subarrays. Make groups of δ of these subarrays and apply the procedure SEGREGATE on all groups in parallel. The elements of each group now appear in a sequence of (not necessarily sorted) subarrays of size at most δ^2 each, so that all elements of a subarray are larger than every element of a preceding subarray; sort these subarrays in parallel using algorithm $\mathcal{A}$, and each group would be sorted. That is, now the input elements are arranged in $\frac{n}{\delta^3}$ sorted subarrays of size δ^3 each. Repeat this process of combining δ sorted subarrays at a time until a single sorted array is left.

For an appropriate choice of δ, this scheme gives the desired time bound.

We now give the construction scheme more formally. The basic building block of the scheme is Valiant's algorithm that merges two arrays of a total size n in $O(\log\log n)$ time with n processors on a CREW PRAM [96, 55]; let a be the constant hidden by the O-notation here. Guided by a complete binary tree on n leaves,

---

Procedure CREW_SSORT$_h(I, n)$

begin

$\delta \leftarrow (\sqrt{\log n})^{1/\log^{(h+2)} n}$;

for $1 \leq i \leq n/\delta^2$ in parallel do /* $I_i^0 = I[(i-1)\delta^2 + 1, \ldots, i\delta^2]$ */

CREW_SSORT$_{h-1}(I_i^0, \delta^2)$;

for $j \leftarrow 1$ to $(\log n / \log \delta) - 2$ do

begin

for $1 \leq i \leq n/\delta^{j+2}$ in parallel do

begin

SEGREGATE$(I_i^j, \delta^{j+2}, \delta)$;

for $1 \leq k \leq \delta^{j+1}$ in parallel do

CREW_SSORT$_{h-1}(G_k(I_i^j), g_k(I_i^j))$;

end

end

end

---

proceeding bottom up, and using Valiant's algorithm at each level, the standard merge sort thus runs in $a \log n \log\log n$ time on an n processor CREW PRAM [55]. This is the first algorithm in our sequence. We denote this algorithm by CREW_SSORT$_0(I, n)$.

For $h > 0$, CREW_SSORT$_h(I, n)$ is given above. Let I_i^j be $I[(i-1)\delta^{j+2} + 1, \ldots, i\delta^{j+2}]$, the i-th subarray of size δ^{j+2} in I; $0 \leq j \leq (\log n / \log \delta) - 2$, and $1 \leq i \leq n/\delta^{j+2}$. Note that $I_{(i-1)\delta+1}^{j-1}, \ldots, I_{i\delta}^{j-1}$ constitute I_i^j.

On a CREW PRAM, PREFIX(n, n, b) is $O(\log n)$, MERGE(n, n) is $O(\log\log n)$ [55], and SEARCH$(\delta, 1)$ is $O(\log \delta)$. Thus, by Lemma 3.1, with one processor per input element, the procedure call "SEGREGATE$(I_i^j, \delta^{j+2}, \delta)$" takes $O(\log \delta + \log\log n)$ time on a CREW PRAM. If $S_h(n)$ is the time taken by CREW_SSORT$_h(I, n)$ with n processors, then

$$S_h(n) \leq O(1) + S_{h-1}(\delta^2) + \left(\frac{\log n}{\log \delta} - 2\right)(O(\log \delta + a \log\log n) + S_{h-1}(\delta^2))$$

Since on a CREW PRAM, with any number of processors, $\Omega(\log\delta)$ is a lower bound on sorting δ^2 elements [23], $\log\delta = O(S_{h-1}(\delta^2))$. Thus, for an appropriate constant b,

$$S_h(n) \leq ab \cdot \frac{\log n}{\log\delta} \cdot \log\log n + b \cdot \frac{\log n}{\log\delta} \cdot S_{h-1}(\delta^2)$$

We claim that,

$$S_h(n) \leq \frac{(2b)^h(4b-1)-2b}{2b-1} \cdot a \log n \log^{(h+2)} n$$

The proof is through induction. When $h = 0$, CREW_SSORT$_0(I,n)$ runs in $a\log n\log\log n$ time; this is the basis. Suppose the claim is true for $h = H-1$. Then,

$$\begin{aligned} S_H(n) &\leq ab \cdot \frac{\log n}{\log\delta} \cdot \log\log n + b \cdot \frac{\log n}{\log\delta} \cdot S_{H-1}(\delta^2) \\ &\leq ab \cdot \frac{\log n}{\log\delta} \cdot \log\log n + b \cdot \frac{\log n}{\log\delta} \cdot \frac{(2b)^{H-1}(4b-1)-2b}{2b-1} \cdot a \log\delta^2 \log^{(H+1)} \delta^2 \end{aligned}$$

Since $\log\delta = \frac{\log\log n}{2\log^{(H+2)} n}$,

$$\begin{aligned} S_H(n) &\leq 2ab \cdot \left(1 + \frac{(2b)^{H-1}(4b-1)-2b}{2b-1}\right) \log n \log^{(H+2)} n \\ &= \frac{(2b)^H(4b-1)-2b}{2b-1} \cdot a \log n \log^{(H+2)} n \end{aligned}$$

Thus we have the following theorem:

Theorem 3.1 *The procedure* CREW_SSORT$_{\log^* n-2}(I,n)$ *sorts an array I of size n, with n processors, on a CREW PRAM, in* $\log n \cdot 2^{O(\log^* n)}$ *time.*

On a CREW PRAM, PREFIX$(n, \frac{n}{\log n}, b) = O(\log n)$, MERGE$(n, \frac{n}{\log\log n}) = O(\log\log n)$ [55], and SEARCH$(\delta, 1) = O(\log\delta)$. The standard optimal merge sort runs in $O(\log n\log\log n)$ time on an $\frac{n}{\log\log n}$ processor CREW PRAM [55]. Thus, by Lemma 3.1, the procedure call "SEGREGATE$(I_i^j, \delta^{j+2}, \delta)$" takes $O(\log\delta + \log\log n + \frac{n}{p})$ time with $p \leq n$ processors on a CREW PRAM. Hence, we have the following theorem:

Theorem 3.2 *The procedure* CREW_SSORT$_r(I,n)$ *sorts an array I of size n, with* $\frac{n}{\log^{(r+2)} n}$ *processors, on a CREW PRAM, in* $O(\log n \cdot \log^{(r+2)} n)$ *time, for a constant integer* $r \geq 0$.

Once again, assume that n processors are available; then the entire sequence of the above algorithms can be captured in the following recursive structure.

Procedure CREW_RSORT(I, n, δ)

begin

if $n \leq \delta^2$ or $\delta = 0$ then

$\delta \leftarrow \sqrt{\log n}$;

if $n = 2$ then

sort I with 2 processors in $O(1)$ time, exit;

Divide the input I into subarrays $I_1, \ldots, I_\delta$ of size n/δ each;

for $1 \leq i \leq \delta$ in parallel do /* recursively sort the subarrays */

CREW_RSORT$(I_i, \frac{n}{\delta}, \delta)$;

SEGREGATE(I, n, δ);

for $1 \leq k \leq n/\delta$ in parallel do

CREW_RSORT$(G_k(I), g_k(I), 0)$;

end

The time taken by a call "CREW_RSORT(I, n, δ)" is

$$S(n, \delta) = O(1) + S\left(\frac{n}{\delta}, \delta\right) + O(\log \delta + \log\log n) + S(\delta^2, 0)$$

Thus, for an appropriate constant b,

$$S(n, \delta) \leq S\left(\frac{n}{\delta}, \delta\right) + b(\log\log n + S(\delta^2, 0))$$

Unrolling the first term,

$$\begin{aligned} S(n, \delta) &\leq S\left(\frac{n}{\delta^2}, \delta\right) + 2b(\log\log n + S(\delta^2, 0)) \\ &= S\left(\frac{n}{\delta^r}, \delta\right) + rb(\log\log n + S(\delta^r, 0)), \qquad \text{for } r > 2 \\ &= S(\delta^2, \delta) + \left(\frac{\log n}{\log \delta} - 2\right) b(\log\log n + S(\delta^2, 0)), \quad \text{with } r = \left(\frac{\log n}{\log \delta} - 2\right) \end{aligned}$$

From the first statement of the procedure, $S(\delta^2, \delta) = S(\delta^2, 0)$. Thus,

$$\begin{aligned} S(n, \delta) &\leq S(\delta^2, 0) + b\left(\frac{\log n}{\log \delta} - 2\right)(\log\log n + S(\delta^2, 0)) \\ &\leq b \cdot \frac{\log n}{\log \delta} \cdot \log\log n + b \cdot \frac{\log n}{\log \delta} \cdot S(\delta^2, 0) \end{aligned}$$

To sort an array I of size n, call "CREW_RSORT$(I, n, 0)$". The time required, with n processors, is

$$S(n,0) = S(n, \sqrt{\log n}) \leq 2b \log n + 2b \cdot \frac{\log n}{\log\log n} \cdot S(\log n, 0)$$

Unrolling this recurrence relation, we get, for $r \geq 1$,

$$S(n,0) = \left(\sum_{i=1}^{r} (2b)^i\right) \log n + \frac{(2b)^r \log n}{\log^{(r+1)} n} \cdot S(\log^{(r)} n, 0) = \frac{2^{O(r)} \log n}{\log^{(r+1)} n} \cdot S(\log^{(r)} n, 0)$$

Thus, $S(n,0) = 2^{O(\log^* n)} \log n$; the procedure call "CREW_RSORT$(I, n, 0)$" is equivalent to the procedure call "CREW_SSORT$_{\log^* n - 2}(I, n)$".

Moreover, since $\log^{(r)} n$ elements can be sorted in $O(\log^{(r+1)} n)$ time with $(\log^{(r)} n)^2$ processors on a CREW PRAM [77], we have the following theorem:

Theorem 3.3 *The procedure call "CREW_RSORT$(I, n, 0)$", with the recursion terminated after $r = O(1)$ levels, sorts an array I of size n in $O(\log n)$ time with $n \log^{(r)} n$ processors on a CREW PRAM.*

3.3 The Algorithm on the Parallel Comparison Model

A generalisation of the CREW PRAM algorithm, given in the previous section, gives the recursive procedure CMP_RSORT(I, n, δ) for the parallel comparison model; the generalisation is only in the choice of δ. To sort an array I of size n, call "CMP_RSORT$(I, n, 0)$". We shall show, later in this section, that the time taken by this call, with $n\alpha$ processors, is $\frac{\log n}{\log \alpha} \cdot 2^{O(\log^*(\log n / \log \alpha))}$

On the parallel comparison model, since $\textsc{Merge}(n, n\alpha) = O(\log(\frac{\log n}{\log \alpha}))$, and $\textsc{Search}(n, p) = O(\frac{\log n}{\log p})$, by Lemma 3.1, the procedure call "$\textsc{Segregate}(I, n, \delta)$" takes $O(\log(\frac{\log n}{\log \alpha}) + \frac{\log \delta}{\log \alpha})$ time with $n\alpha$ processors.

Following the analysis of CREW_RSORT(I, n, δ) in the previous section, since $\Omega(\frac{\log \delta}{\log \alpha})$ is the lower bound on sorting δ elements with $\delta\alpha$ processors, the time taken by a procedure call "CMP_RSORT(I, n, δ)" is, with $n\alpha$ processors, for an appropriate constant b,

$$\begin{aligned} S(n,\delta) &\leq S\left(\frac{n}{\delta}, \delta\right) + b \log\left(\frac{\log n}{\log \alpha}\right) + b \cdot S(\delta^2, 0)) \\ &\leq b \cdot \frac{\log n}{\log \delta} \cdot \log\left(\frac{\log n}{\log \alpha}\right) + b \cdot \frac{\log n}{\log \delta} \cdot S(\delta^2, 0) \end{aligned}$$

Procedure CMP_RSORT(I, n, δ)

begin

if $n \leq \delta^2$ or $\delta = 0$ then

$\delta \leftarrow \sqrt{\alpha^{\log(\log n/\log \alpha)}}$;

if $n = \alpha$ then

sort I with α^2 processors in one step, exit;

Divide the input I into subarrays $I_1, \ldots, I_\delta$ of size n/δ each;

for $1 \leq i \leq \delta$ in parallel do /* recursively sort the subarrays */

CMP_RSORT$(I_j, n/\delta, \delta)$;

SEGREGATE(I, n, δ);

for $1 \leq k \leq n/\delta$ in parallel do

CMP_RSORT$(G_k(I), g_k(I), 0)$;

end

To sort an array I of size n, call "CMP_RSORT$(I, n, 0)$". The time required, with n processors, is

$$\begin{aligned} S(n,0) &= S(n, \sqrt{\alpha^{\log(\log n/\log\alpha)}}) \\ &\leq 2b \cdot \frac{\log n}{\log \alpha} + 2b \cdot \frac{\log n/\log\alpha}{\log(\log n/\log\alpha)} \cdot S(\alpha^{\log(\log n/\log\alpha)}, 0) \end{aligned}$$

Unrolling this recurrence relation, we get, for $r \geq 1$,

$$S(n,0) = \left(\sum_{i=1}^{r} (2b)^i\right) \frac{\log n}{\log \alpha} + \frac{(2b)^r \log n/\log\alpha}{\log^{(r)}(\log n/\log\alpha)} \cdot S(\alpha^{\log^{(r)}(\log n/\log\alpha)}, 0)$$

With $r = \log^*(\log n/\log\alpha)$, $S(n,0) = 2^{O(\log^*(\log n/\log\alpha))} \cdot \frac{\log n}{\log\alpha} \cdot S(\alpha, 0)$. On the parallel comparison model, α elements can be sorted in unit time with α^2 processors: compare every element with every other element. That is, $S(\alpha, 0) = 1$. Thus, $S(n,0) = 2^{O(\log^*(\log n/\log\alpha))} \cdot \frac{\log n}{\log\alpha}$.

Theorem 3.4 *The procedure call "CMP_RSORT$(I, n, 0)$" sorts an array I of size n in $2^{O(\log^*(\log n/\log\alpha))} \cdot \frac{\log n}{\log\alpha}$ time with $n\alpha$ processors on the parallel comparison model.*

3.4 The Algorithm on a CRCW PRAM

The CRCW PRAM implementation of the algorithm is similar to the parallel comparison model version, except in that when $n = \alpha$, with α^2 processors, sorting would require $O(\frac{\log\alpha}{\log\log\alpha})$ time instead of $O(1)$ time. On a CRCW PRAM,

$$\textsc{Prefix}(n,p,b) = O\left(\frac{\log n}{\log\log p} + \min\left\{\log\frac{b}{\log p}, \log n\right\}\right)$$

[91, 41], $\textsc{Merge}(n, n\alpha) = O(\log(\frac{\log n}{\log\alpha}))$, and $\textsc{Search}(n,p) = O(\frac{\log n}{\log p})$. Thus, by Lemma 3.1, the procedure call "$\textsc{Segregate}(I,n,\delta)$" takes $O(\frac{\log\delta}{\log\log\delta\alpha} + \frac{\log\delta}{\log\alpha} + \log(\frac{\log m}{\log\alpha}))$ time with $n\alpha$ processors on a CRCW PRAM. Since, $\Omega(\frac{\log\delta}{\log\log\delta\alpha} + \frac{\log\delta}{\log\alpha})$ is the lower bound on sorting δ elements with $\delta\alpha$ processors, it is clear that the recurrence relation

$$S(n,0) = 2b\cdot\frac{\log n}{\log\alpha} + 2b\cdot\frac{\log n/\log\alpha}{\log(\log n/\log\alpha)}\cdot S(\alpha^{\log(\log n/\log\alpha)},0)$$

holds for CRCW PRAM too, but with a larger value of b than for the comparison model version. Solving this, we have the following result:

Theorem 3.5 *The procedure call "*CMP_RSORT$(I,n,0)$*" for sorting an array I of size n can be implemented in $2^{O(\log^*(\log n/\log\alpha))}\cdot\frac{\log n}{\log\log\alpha}$ time with $n\alpha$ processors on a CRCW PRAM.*

For the CRCW PRAM model, Beame and Hastad have shown an $\Omega(\frac{\log n}{\log\log n\alpha})$ lower bound on sorting with $n\alpha$ processors [10]. Coupled with the comparison model lower bound, thus, sorting requires $\Omega\left(\frac{\log n}{\log\alpha} + \frac{\log n}{\log\log n\alpha}\right)$ time on a CRCW PRAM. But it is not known whether this lower bound is tight. Cole's merge sort runs on a CRCW PRAM in $O(\frac{\log n}{\log\log\alpha})$ time. For a while this was believed to be the best possible. But off late, entirely different techniques [45, 35] have suggested faster algorithms, though still not matching the known lower bound. Our algorithm, though slower, could be deemed simpler than these algorithms.

Chapter 4

Optimal Sublogarithmic Time 3-Colouring of Rooted Forests

Being a bipartite graph, a rooted forest is 2-colourable. But it would take $\Omega(\frac{\log n}{\log\log n})$ time to 2-colour even a linked list, even with as many as a polynomial number of processors [32]; besides, there is evidence to suggest that this lower bound could be strengthened to $\Omega(\log n)$ [90]. On the other hand, a rooted forest can be 3-coloured in sublogarithmic time: Goldberg et.al. [32] and Han [48] describe an $O(\log^* n)$ and an $O(\log(\log^* n))$ time algorithm respectively—both algorithms run on an n processor CREW PRAM; these algorithms are not optimal. Attempts to find an optimal parallel sublogarithmic time algorithm for 3-colouring have been successful only for rooted forests with a constant maximum degree [84, 87]. The algorithm of [84] takes $O(\frac{\log n}{\log\log n})$ time on a CRCW PRAM; the time bound was improved to $O(\log^{(k)} n)$ on an EREW PRAM in [87]. In this chapter, we present an $O((\log\log n)\log^*(\log^* n))$ time optimal algorithm on the TOLERANT CRCW PRAM, for 3-colouring general rooted forests.

The fractional independent set (FIS henceforth) problem is to find an independent set that constitutes at least a constant fraction of the vertices. We show that given an $f(n)$ time algorithm for 3-colouring a rooted forest F, an FIS of F can be found in $O(f(n))$ time with the same number of processors, on both COMMON and TOLERANT CRCW PRAMs; in other words, the FIS problem is no harder than 3-colouring. Furthermore, with randomisation, an FIS of a rooted forest can be found in $O(1)$ time on an n processor COMMON or TOLERANT CRCW PRAM. We use this result to obtain an optimal algorithm for 3-colouring a rooted forest with high

probability, in $O((\log^* n)\log^*(\log^* n))$ time.

On rooted forests, top-down algebraic computation problems like 2-colouring, depth computation for all vertices, and prefix summation can be solved using pointer jumping in $O(\log n)$ time with n processors on a CREW PRAM. If the input forest is either known to be a linked list [6], or given in adjacency list representation [1], then all these problems can be solved in $O(\log n)$ time optimally on an EREW PRAM. On the other hand, when each vertex knows only its parent, the known logarithmic time optimal algorithm [49] can at best be implemented on a COLLISION CRCW PRAM. The 3-colouring algorithm proposed here suggests optimal $O(\log n)$ time algorithms for these problems on a much weaker model—TOLERANT. On a COMMON CRCW PRAM, we show, these problems can be solved in $O(\log n)$ time with a time-processor product of $O(n\log(\log^* n))$.

To obtain the above algorithms on TOLERANT, we show that a TOLERANT PRAM of size N with a linear address space, can be slowed down by any factor $\lambda = \Omega(\log\log N)$, with no asymptotic increase in space or cost. This is a non-trivial result since TOLERANT is not known to be self-simulating in general [39]. Also, this result complements a corresponding observation in [38] that concerns factors $\lambda = O(1)$.

4.1 Preliminaries

A PRAM model is said to be *self-simulating* , if for all $N \geq n \geq 1$, a PRAM of that model of size n can simulate a single step of another PRAM of the same model of size N in $O(\frac{N}{n})$ time. All CRCW PRAMs that are at least as powerful as COLLISION or COMMON are self-simulating. TOLERANT is not known to be self-simulating [39] (see also [4]).

The *approximate prefix summation* (APS) problem over a sequence $\langle x_1, \ldots, x_n\rangle$ of non-negative integers is to find a a sequence $\langle 0 = y_0, y_1, \ldots, y_n\rangle$ so that for every $1 \leq i \leq n$, $y_i = \Theta(\sum_{j=1}^{i} x_j)$ and $y_i - y_{i-1} \geq x_i$. The *linear approximate compaction* (LAC) problem over an array of n cells, at most m of which contain an item, the others being empty, is to arrange all the items into an array of size $O(m)$. In ordered LAC, the output should have the same relative ordering of items as the input. Clearly, APS can be used to solve ordered LAC. Goldberg and Zwick [35] show that APS (and hence ordered LAC) can be solved in $O(\log\log n)$ time using $\frac{n}{\log\log n}$ processors on a COM-

MON CRCW PRAM; a matching lower bound is also known [15]. Using the $O(1)$ time n^2 processor simulation of COMMON on TOLERANT [66], it is easy to see that their algorithm can be extended to TOLERANT. Thus, with $p \leq \frac{n}{\log\log n}$ processors, on a TOLERANT PRAM, APS can be solved in $O(\frac{n}{p} + \log\log n)$ time: divide the instance into p groups of equal size, find the prefix sums in each sequentially, and finally combine the sums using a global APS.

We define "an inverse" H of the logstar function as follows: $H(0) = 1$ and $H(i) = 2^{H(i-1)}$, for $i \geq 1$; if $H(x) = n$ then $\log^* n = x$.

4.2 Designing Optimal Algorithms

We say, a problem $\mathcal{P}$ is *ϵ-contractable*, $0 < \epsilon < 1$, if for every instance X of $\mathcal{P}$ of size n, there exists an instance Y of $\mathcal{P}$ of size ϵn, such that Y can be constructed from X, and a solution to X can be found from a solution to Y.

Theorem 4.1 (cf. [40, 30, 31]) *Suppose linear approximate compaction (LAC) of size n has an $O(\frac{n}{p} + C(n))$ time algorithm with p processors. Let $\mathcal{P}$ be a problem that can be solved in $T(n)$ time with n processors, and can be $\frac{1}{2}$-contracted in $O(\frac{n}{p} + R(n))$ time with p processors. If the model used is self-simulating, then $\mathcal{P}$ can also be solved in $O(\frac{n}{p} + R(n)\log T(n) + C(n)\log^* T(n) + T(n))$ time with p processors.*

Proof: To prove the lemma, it is enough to show that $\mathcal{P}$ can be $\frac{1}{T(n)}$-contracted in $O(\frac{n}{p} + R(n)\log T(n) + C(n)\log^* T(n))$ time with p processors, since, we can then solve the contracted instance in $O(T(n))$ time with $\frac{n}{T(n)}$ processors using the given algorithm. Starting with the given instance of size n, proceed in phases. In the i-th phase we perform at least an $\left(H^{(i)}(0)/H^{(i+1)}(0)\right)$-contraction followed by an LAC of the array containing the contracted instance. Thus, the size of the array at the beginning of phase i is at most $A_i = n/H^{(i)}(0)$.

The contraction in the i-th phase proceeds in $H^{(i)}(0)$ stages, the j-th stage of which performs $H(j-1)$ consecutive $\frac{1}{2}$-contractions followed by an LAC of the array containing the contracted instance. That is, the j-th stage performs a $\frac{1}{H(j)}$-contraction. Clearly, the size of the array at the beginning of stage j is at most $B_{i,j} = A_i/\prod_{k=0}^{j-1} H(k)$. Hence, the size of the array after $H^{(i)}(0)$ stages is is at most $A_i/\prod_{k=0}^{H^{(i)}(0)} H(k) < A_i/H^{(i+1)}(0) < A_{i+1}$. The time taken by stage j is $O(H(j-1)(\frac{B_{i,j}}{p} + R(n)) + C(n))$.

Summing the time taken, over all stages, the i-th phase takes

$$O\left(\frac{A_i}{p} + R(n)H(H^{(i)}(0)-1) + C(n)H^{(i)}(0)\right)$$
$$= O\left(\frac{A_i}{p} + R(n)\log H^{(i+1)}(0) + C(n)\log^* H^{(i+1)}(0)\right)$$

time. Note that $H(H^{(i)}(0)-1) = \log H^{(i+1)}(0)$. After the last phase, say the r-th one, the array size is at most $\frac{n}{T(n)}$. Hence, $H^{(r+1)}(0) = T(n)$. So, summing the time taken over all phases, we get the claimed time bound. Hence the lemma. □

TOLERANT, unlike other standard PRAM models [39], is not known to be self-simulating. Thus, Theorem 4.1 cannot be applied on TOLERANT directly. But, we show that when the memory used is linear in the number of processors, TOLERANT is self-simulating. That is, when both the given linear processor algorithm, and the $\frac{1}{2}$-contraction routine can be executed on TOLERANT with linear space, Theorem 4.1 can be applied on TOLERANT as well. The following theorem states the self-simulating property more formally:

Theorem 4.2 *One step of a TOLERANT CRCW PRAM of size N that has an address space of size $O(N)$ can be simulated on another TOLERANT CRCW PRAM of size n in $O(\frac{N}{n})$ time where $n \leq \frac{N}{\log\log N}$, without any asymptotic increase in memory.*

For a discussion on earlier results, as well as a proof of this theorem see Section 4.5.

4.3 Optimal 3-colouring of rooted-forests in sublogarithmic time

Our algorithm is based on the observation that in every rooted forest $F = (V, E)$ at least half the vertices have at most one child; let Z be the set of these vertices. On a TOLERANT CRCW PRAM, the vertices in Z can be identified in $O(1)$ time, with $2n$ processors: If a node v and all its children simultaneously write in v and the write succeeds, then v is a leaf. On the other hand, if all children of v simultaneously write in v and the write succeeds, then v has exactly one child. Remove the vertices of Z from the forest $F[V]$; the resulting forest is $F[V - Z]$.

Since $F[Z]$, the forest induced by the removed vertices, is a collection of linked lists, we can find a 3-colouring $\sigma : Z \to \{1,2,3\}$ of $F[Z]$ in $O(\frac{n}{p} + \log^{(k)} n)$ time with p

processors [48]. Using the colouring σ, the given 3-colouring $\phi : (V - Z) \rightarrow \{1,2,3\}$ of the remaining forest $F[V - Z]$ can be extended to the original forest $F[V]$:

for each $x \in \{1,2,3\}$, in turn,
  for each removed vertex z with $\sigma(z) = x$,
    set $\phi(z)$ to the minimum colour not present in
    z's neighbourhood in the original forest $F[V]$.

Since each removed vertex z has at most two neighbours in the original forest, such a colour can always be found from $\{1,2,3\}$.

Thus the problem of 3-colouring a rooted forest can be $\frac{1}{2}$-contracted in $O(\frac{n}{p} + \log^{(k)} n)$ time using p processors on a TOLERANT CRCW PRAM. Note that the address space used is of size $O(n)$ only. Hence, from Theorem 4.1 and Theorem 4.2, we have the following result:

Theorem 4.3 *A rooted forest can be optimally 3-coloured in $O((\log\log n)\log^*(\log^* n))$ time on a TOLERANT CRCW PRAM.*

Corollary 4.1 *A fractional independent set of a rooted forest can be found in $O(\frac{\log n}{\log\log n})$ time optimally on a TOLERANT CRCW PRAM.*

Proof: Given a 3-colouring of a rooted forest F, a three-fold counting of the vertices can give the colour that is used the largest number of times; this colour is used by at least one third of the vertices. Use the $O(\frac{\log n}{\log\log n})$ time optimal prefix sums algorithm [22] for counting. □

With the help of randomness, both the FIS and 3-colouring problems on rooted forests can be solved much faster.

Lemma 4.1 *A fractional independent set of a rooted forest can be found with high probability, in $O(1)$ time on a COMMON or TOLERANT CRCW PRAM, with n processors.*

Proof: Let a set $\mathcal{I}$ be initialised to the set of leaves in $F = (V, E)$, the given forest. Each vertex in $\mathcal{I}$ informs its parent that it is in $\mathcal{I}$ using a concurrent write. For each vertex v of F, "toss" a fair coin; let the result of the toss $t(v)$ be 0 or 1 randomly with equal probability. For every vertex v which is not a root, if v does not have a child in $\mathcal{I}$, and $t(v) = 1$, and $t(p(v)) = 0$, then add v to $\mathcal{I}$.

Let V_1 be the set of vertices with exactly one child in F and say, $| V_1 | = n_1$. That is, $V - V_1$ consists of the leaves as well as the vertices with two or more children. So, at least a constant fraction of $V - V_1$ are leaves, and hence, if $n_1 = o(n)$, $\mathcal{I}$ is an FIS. On the other hand when $n_1 = \Theta(n)$, at least a constant fraction of the vertices in V_1 have been added to $\mathcal{I}$, with high probability [55]. Hence the lemma. □

Since, every vertex in Z has at most one child in $F[V]$, a 3-colouring of $F[V - Z]$ can be easily extended to $F[V]$. As there is an optimal randomized Las Vegas algorithm for LAC that runs in $O(\log^* n)$ time with high probability [31, 37], applying Theorem 4.1, we have:

Theorem 4.4 *There is an optimal randomized $O((\log^* n)\log^*(\log^* n))$ time algorithm for 3-colouring a rooted forest on a CRCW PRAM.*

In the rest of this section, we show how to use the sublogarithmic time optimal 3-colouring algorithm to get $O(\log n)$ time optimal algorithms for decomposable top-down algebraic computation problems (henceforth referred to as DTAC) on rooted forests, like 2-colouring, depth computation, and prefix summation.

An instance of size n of the *top-down algebraic computation* problem [1] can be represented by a triplet $\langle S, \mathcal{F}, F \rangle$, where S is a set, $\mathcal{F}$ is a set of functions from S to S, and $F = (V, E)$ is a forest on n vertices in which each vertex v has a unique label l_v such that, $l_v \in S$ when v is a root and $l_v \in \mathcal{F}$ otherwise. The problem is to compute a function $\delta : V \rightarrow S$, which can be recursively defined as follows: for $v \in V$, $\delta(v) = l_v$ if v is a root, and $\delta(v) = l_v(\delta(p(v)))$ otherwise.

A top-down algebraic computation is said to be *decomposable* [1] if (i) each $f \in \mathcal{F}$ is computable, (ii) given an algorithm $N(f)$ that computes $f \in \mathcal{F}$, and $s \in S$, $f(s)$ can be computed in $O(1)$ time sequentially, (iii) for each $f \in \mathcal{F}$, each vertex of the input forest that is labelled by f can compute $N(f)$ in $O(1)$ time, (iv) $\mathcal{F}$ is closed under composition, and (v) for $f_1, f_2 \in \mathcal{F}$, $N(f_1 \circ f_2)$ can be computed in $O(1)$ time sequentially.

When $S = \{0, 1\}$ and $\mathcal{F}$ contains only the Boolean negation operation, DTAC reduces to *2-colouring*. In *Ranking* or *depth computation* S is the set of integers, the roots in the input forest are labelled by 1, and $\mathcal{F}$ contains only the increment function. When S is the set of integers and $\mathcal{F} = \{f_i \mid f_i(x) = x + i;\ x, i \in S\}$, DTAC reduces to prefix summation.

Given an instance $\langle S, \mathcal{F}, F \rangle$ of DTAC of size n, where F is a rooted forest, find an FIS Z of F using Corollary 4.1. For each $z \in Z$, remove z from F, and for each child w of z, set $p(w) = p(z)$ and $l_w = l_w \circ l_z$. Let us call the resultant rooted forest F'. By the definition of DTAC, once Z has been found, the construction of F' can be done in $O(1)$ time with n processors, without concurrent writes.

Assume that the solution $\delta' : (V - Z) \rightarrow S$ for $\langle S, \mathcal{F}, F' \rangle$ has been found. The solution for the original DTAC $\delta : V \rightarrow S$ can be obtained from δ' by the following computation: $\delta(v) = l_v$ if $v \in Z$ and v is a root in F; $\delta(v) = l_v(\delta(p(v)))$ if $v \in Z$ and v is not a root in F; and $\delta(v) = \delta'(v)$ otherwise. By the definition of DTAC, again, this computation would take $O(1)$ time with n processors, without concurrent writes.

Thus we have that DTAC can be $\frac{2}{3}$-contracted in $O(\frac{\log n}{\log \log n})$ time with $O(n)$ space and $O(n)$ operations on a TOLERANT CRCW PRAM.

Hence, from Theorem 4.1 and Theorem 4.2 we have the following result:

Theorem 4.5 *An instance of DTAC of size n can be solved on a rooted forest in $O(\log n)$ time with $\frac{n}{\log n}$ processors, on a TOLERANT CRCW PRAM.*

4.4 A deterministic algorithm for the fractional independent set

In this section we show that given an $f(n)$ time algorithm for 3-colouring a rooted forest $F = (V, E)$, an FIS of F can be found in $O(f(n))$ time with the same number of processors, on both COMMON and TOLERANT CRCW PRAMs; in other words, the FIS problem is no harder than 3-colouring. Instead of counting the number of vertices of each colour to find the largest colour class in a 3-colouring of F, we recolour some of the vertices so as to ensure that at least one-fourth of the vertices are of colour 1. This is done by guaranteeing that each vertex not of colour 1 has a near enough descendant of colour 1. The method is described in more detail below:

Let L be the set of leaves in F; remove all leaves from F after giving them colour 1 (they are never to be recoloured). Let $F_L = F[V - L]$ be the resultant forest. Obtain a 3-colouring of F_L.

Every root of colour 1 (2 or 3) assumes that it has a parent, grand parent and great grand parent of colours 2, 1 and 2 (1, 2 and 1) respectively. Note that we are

not really adding any new vertex to F_L. Recolour each vertex in F_L with the colour of its great-grand parent, real or assumed. Now, every vertex is coloured the same as its siblings.

Starting at a child of a vertex of colour 3, and going towards the root, the possible prefixes (of size three) of colour sequences encountered are 1-3-2, 2-3-1, 1-3-1 and 2-3-2. We now affect a recolouring so that only instances of 1-3-2 remain in F_L. First, sequences 1-3-1 and 2-3-2 are removed by recolouring the vertex of colour 3 (the middle one) with colour 2 and colour 1 respectively. Now, the only possible extensions, terminating at a colour 2 vertex, for a 2-3-1 sequence are 2-3-1-3-2 and 2-3-1-2. Recolour these into sequences 2-1-2-1-2 and 2-1-3-2 respectively. Note that recolouring these sequences does not require a concurrent write, if each vertex remembers the very first colouring received by itself and a few of its immediate ancestors. Every vertex of colour 3 now has a parent of colour 2 and children of colour 1. A vertex of colour 2 that does not have a child of colour 1, has all its children coloured 3, and hence surely has a grandchild of colour 1. So, every vertex of F_L is either coloured 1 or has a child or a grand-child coloured 1.

Reinsert L into F_L. If there are colour conflicts, then recolour the parent in each conflict with colour 4. Every vertex of F is now either coloured 1 or has a child or a grand-child or a great-grand child of colour 1. At least one fourth of the vertices in F are now coloured 1.

The power of concurrent write has been used only to identify the leaves as well as to locate colour conflicts. Hence, we have the following theorem:

Theorem 4.6 *If a rooted forest F can be 3-coloured in $f(n)$ time on a CRCW PRAM model, then a fractional independent set of F can be found in $O(f(n))$ time on the same model, provided the model is powerful enough to compute the OR of n bits in $O(1)$ time with $O(n)$ operations.*

Thus, using the algorithm of [48] for 3-colouring, DTAC can be $\frac{3}{4}$-contracted in $O(\log(\log^* n))$ time with n processors on a COMMON CRCW PRAM.

Hence, using Theorem 4.1 with minor modifications, we have the following theorem:

Theorem 4.7 *An instance of DTAC of size n, can be solved on a rooted forest in $O(\log n)$ time with $O(n \log(\log^* n))$ operations on a COMMON CRCW PRAM.*

4.5 The Self-Simulation Property of TOLERANT CRCW PRAM

In this section, we present the proof of Theorem 4.2, which essentially states that a TOLERANT PRAM of size N and memory $O(N)$ can be slowed down, by a factor of $\lambda = \Omega(\log\log N)$, without any cost or space penalties. A corresponding randomised result is known: Gil, Matias and Vishkin [31] have shown that a TOLERANT PRAM of size N and memory $O(s)$ can be slowed down, with high probability, by a factor of $\lambda = \Omega(\log^* N)$ without a cost penalty but with an $O(\frac{N(\log N)(\log\lambda)}{\log(\log^* N)})$ additive overhead in space. Note that our result is limited to TOLERANT PRAMs of linear space and to $\lambda = \Omega(\log\log N)$, but takes only $O(N)$ space. On the other hand, Grolmusz and Ragde [38] have shown the self-simulation property of a linear space TOLERANT PRAM for $\lambda = O(1)$.

Let "Γ_N" denote "a TOLERANT PRAM of N processors," in the sequel.

Lemma 4.2 *One step of Γ_N that has an address space of size s can be simulated on Γ_n in $O(\frac{N}{n})$ time, when $s, n \leq N$ and $n = 2s$.*

Proof: If every processor of Γ_N that is in a conflict were to back out from writing, then the remaining writes can be self-simulated as in a CREW PRAM, in $O(\frac{N}{n})$ time. We use a flag for each memory cell to signify conflicts. The flag of a cell would take on values 0, 1 or greater than 1, depending on whether 0, 1 or more processors of Γ_N want to write in that cell. Initially, all flags are set to 0.

Divide Γ_n into two equal halves: the guards and the actors. Each guard stands by a unique memory cell. Also, divide Γ_N into $g = \lceil\frac{2N}{n}\rceil$ groups of size $\frac{n}{2}$ each. Activate the groups one by one, in g rounds, using the actors; the i-th actor deputes for the i-th processor activated in the current round and is associated with the cell that this processor wants to write into. In each round, the actors and guards together can be viewed as forming a rooted forest, where each guard is a root and the actors associated with it are its children. Whether a root (guard) has 0, 1 or more children (actors), can be ascertained in $O(1)$ time on a TOLERANT PRAM: if a node v and all its children simultaneously write in v and the write succeeds, then v has no child; on the other hand, if all its children simultaneously write in v and the write succeeds, then v has exactly one child.

For each cell, in every round the guard increments its flag by 0, 1, or 2 depending on whether it has 0, 1, or more children. It is easy to see that at the end of g rounds the flags would correctly reflect the conflict status. The processors can now read the flags and back out if necessary.

Hence the lemma. □

Corollary 4.2 *One step of Γ_N that has an address space of size s can be simulated on Γ_n in $O(\frac{Ns}{n^2})$ time, where $n \leq s, N$.*

Proof: Divide the memory into $\lceil \frac{2s}{n} \rceil$ segments of size n each. Proceed in $\lceil \frac{2s}{n} \rceil$ rounds, in the i-th round simulating the writes into the i-th segment using Lemma 4.2 in $O(\frac{N}{n})$ time. □

The problem of *padded-sorting* n values is to output them in sorted order in an array of size at most $O(n)$; unused cells of the output array are to be filled with some special character [44].

Lemma 4.3 *Given m integers in the range $[1 \cdots r]$ in an array of size m, they can be padded-sorted in $O(\frac{m}{p})$ time with p processors, on a TOLERANT CRCW PRAM, where $r = O(\frac{m}{p})$ and $p \leq \frac{m}{\log \log m}$.*

Proof: We employ a method similar to the one used for bucket-sorting m integers in the range $[1 \cdots \log^{O(1)} m]$ in [22]. The following procedure padded-sorts m integers in the range $[1 \cdots r]$ with the claimed resource bounds.

Step 1: Divide the set of m integers into p groups of size $\lceil \frac{m}{p} \rceil$ each; the last group may be of a smaller size. Assign one processor for each group and sort it sequentially. For a $p \times r$ array C, let $C[i,j]$ be the number of items with key j in group i. Let the first $C[i,j]$ locations of an array $A_{i,j}$ contain the items with key j in group i.

Step 2: Let D be an array of size pr. Let $D[(j-1)p+i] = C[i,j]$, for $1 \leq i \leq p$ and $1 \leq j \leq r$; that is, we copy C into D column by column. Hence, $D[(j-1)p+1], \ldots, D[jp]$ contain the number of items with key j in groups $1, \ldots, p$ respectively.

Step 3: Find the approximate prefix sums over D. For $0 \leq i \leq pr$, let $\alpha(i)$ be the i-th sum. For $1 \leq i \leq p$ and $1 \leq j \leq r$, copy the first $C[i,j]$ locations of $A_{i,j}$ into consecutive cells of a base array of size $\alpha(pr)$, starting at address $\alpha((j-1)p+i-1)+1$.

Time taken is clearly $O(\frac{m}{p})$. The objects in the base array, where there are, now appear in sorted order. □

Lemma 4.4 *One step of Γ_N that has an address space of size N can be simulated on Γ_n in $O(\frac{N}{n})$ time without any asymptotic increase in memory, where $n \leq \frac{N}{\log\log N}$.*

Proof: Let $P_1, \ldots, P_N$ be the processors of Γ_N.

Divide the address space into $\lceil \frac{N}{n} \rceil$ segments of size n each. For $1 \leq l \leq N$, let $S(l)$ be the number of the segment into which P_l wants to write. Form an array of ordered pairs $(S(l), l)$ and padded-sort these on the first key, using Lemma 4.3; with n processors, this takes $O(\frac{N}{n})$ time. Thus, all processors of Γ_N that wish to write in the same segment have now come together in a base array of size $O(N)$.

Consider the segments one by one. Suppose the processors attempting to write in the i-th segment are spanning over N_i cells of the base array. If $N_i \geq n$, applying Corollary 4.2, the write attempts into the i-th segment can be simulated in $O(N_i/n)$ time. When $N_i < n$, with n processors the simulation clearly can be done in $O(1)$ time.

The total time taken for the simulation is $O(\Sigma \frac{N_i}{n}) = O(\frac{N}{n})$. Note that the space used outside of the approximate prefix summation routine is linear. Hence the lemma. □

We can now prove Theorem 4.2, as follows: Let us assume that the address space is of size cN, for some constant c. Divide the memory into segments of size N each. Proceed in c phases; in the i-th phase, use Lemma 4.4 to simulate the processors writing in the i-th segment of the memory. Each phase takes $O(\frac{N}{n})$ time; so the total time taken is $O(\frac{cN}{n}) = O(\frac{N}{n})$. Hence the theorem.

In this proof, when the deterministic algorithm for approximate prefix summation is replaced by the $O(\log^* n)$ time randomised algorithm [31], we get a corresponding randomised result for $n \leq \frac{N}{\log^* N}$. This is an improvement over the earlier self-simulation of linear space TOLERANT [31] in terms of memory used.

Chapter 5

Brooks' Colouring

In this chapter, an $O(\Delta^2 \log \Delta \log n)$ time algorithm for Δ-colouring a Brooks' graph, with $n/\log n$ processors, on an EREW PRAM, is obtained. In particular, this algorithm 3-colours a 3-degree Brooks' graph in $O(\log n)$ time, with $n/\log n$ processors, on an EREW PRAM. (The basic idea of this algorithm, in the context of a CREW PRAM, was treated in the author's M. Tech. thesis [85].) See also [88] This is an improvement over the previous best known algorithms, on bounded degree graphs [58, 80]. (See the discussion in Section 1.2.3.)

We also obtain the following combinatorial result: for any two vertices u and v that are more than a constant distance apart in a 3-degree Brooks' graph G, there exist two distinct 3-colourings of G of which one has u and v coloured the same, and the other has u and v coloured differently. Thus, no vertex in a Brooks' graph can force the colour of vertices arbitrarily far away. This highly local nature of Brooks' colouring can be seen as suggesting that on a CRCW PRAM, the problem may have a $o(\log n)$ time algorithm.

For the problem of Δ-colouring, a colour c is said to be *feasible* at v if v does not have a neighbour coloured c. An uncoloured vertex in a partially coloured graph is said to be *at impasse* if it has no feasible colour in $\{1, \ldots, \Delta\}$. If v is a vertex at impasse, then its neighbour of colour i will be denoted by $v_i (i = 1, \ldots, \Delta)$.

An α-β component in G is a component of the subgraph induced by vertices coloured α or β. Note that interchanging colours α and β in an α-β component does not affect the validity of the colouring. So, for a vertex v at impasse if v_α and v_β belong to different α-β components interchanging colours in one of them will resolve

the impasse.

We define a construct called *fork* as follows: given a partially coloured Brooks' graph $G = (V, E)$, a fork at $v \in V$ is a minimal simple path F of at least two vertices with v as an endpoint, so that the other endpoint $e(F)$ of F either (i) has a degree of at most $\Delta - 1$ in G, or (ii) has a degree of Δ in G, and has two neighbours, both of which are not in F, and are coloured the same. If v is at impasse, and the neighbour of v in F is the α-coloured neighbour v_α of v in G, then we call F an α-fork.

For each vertex w in F, $w \neq e(F)$, let the successor of w, be the farther from v of its neighbours in F.

If v is at impasse, then using F the impasse can be resolved as follows: Recolour each $x \in F$, $x \neq e(F)$, with the colour of its successor in F. Uncolour $e(F)$. Give $e(F)$ one of the colours missing in its neighbourhood; note that, by definition, such a colour should exist.

Similarly, if v is coloured, then using F we can recolour v: Recolour each $x \in F$, $x \neq e(F)$, with the colour of its successor in F. Uncolour $e(F)$. Give $e(F)$ one of the colours missing in its neighbourhood.

5.1 An Algorithm for Brooks' Colouring

First, we present a procedure that will be subsequently used in the algorithm.

Procedure MAXIMAL_SET(Δ, α, β)

Input: A Δ-regular graph $G = (V, E)$, in which every α-β component is a simple path and for every vertex v at impasse v_α and v_β are end points of α-β chains.

Output: A maximal set of α-β components in G, such that no two members in the set *touch* the same impasse vertex; a component Γ touches a vertex v if at least one vertex in Γ is adjacent to v.

Step 1: Form a graph H in which each vertex corresponds to an α-β component of G. Place an edge between two vertices of H if and only if the α-β components corresponding to them touch the same impasse vertex. Since an α-β component can touch at most two impasse vertices, one at each end, the maximum vertex degree of H is 2. Remove all isolated vertices from H.

Step 2: Find the connected components of H and identify each as a chain or a cycle. From each cycle remove a vertex. Let the resulting graph be H'. List rank each component of H'. For every odd ranked vertex of H' interchange colours α and β in the corresponding α-β component of G.

An algorithm for 3-colouring of 3-degree graphs is now presented.

Procedure 3_Colour_Cubic_Graphs

Input: A cubic graph $G = (V, E)$ in adjacency list representation.

Output: A 3-colouring $\sigma : V \rightarrow \{1, 2, 3\}$ of G.

Step 1: Find an MIS M of G, and for each $v \in M$ let $\sigma(v) = 3$. The graph $G - M$ has a maximum degree of 2; that is, $G - M$ consists of disjoint chains and cycles. From every odd cycle of $G - M$ elect a representative into a set I. Clearly, $G - M - I$ is 2-colourable; 2-colour it. Each vertex of I is at impasse, and is left uncoloured.

REMARK: For each vertex $v \in V$, we keep two flags $P(v)$ and $Q(v)$. Each flag can take on values "up" and "down". Initially all flags are down. Every flag that is up, will be associated with an impasse vertex; that is, every flag that is up will hold a pointer to an impasse vertex. As soon as the impasse is resolved at a vertex, all its associated flags will be brought down. We shall use the following procedure extensively.

Procedure RESOLVE

```
begin
    For v ∈ I do in parallel
        if v has at least one colour missing in its neighbourhood then
            give v the minimum feasible colour, remove it from I,
            and bring down every flag associated with v
end.
```

In the sequel, a vertex with its P-flag up will be called a P-vertex, and a component in the subgraph induced by P-vertices a P-component; and, similarly for flag Q.

Step 2: For each odd cycle C of $G - M$, associate the P-flags of all coloured vertices in C with the impasse vertex in C, and put up all of them.

REMARK: Every odd cycle of $G - M$ provides a path between v_1 and v_2, if v is the impasse vertex contained in it. Moreover, each vertex on this path is coloured 1 or 2

and its neighbour outside the path is coloured 3. Thus, v_1 and v_2 belong to the same $1-2$ component, which is a simple path with v_1 and v_2 as its end points.

Step 3: Recolour all colour-1 vertices with colour 2 where feasible. Recolour all colour-3 vertices with colour 2 where feasible. Call RESOLVE.
REMARK: Every $1-3$ component of G is a now simple path. Thus, each of v_3 and v_1 is an end point of a $1-3$ chain. That is every impasse vertex has exactly two (not necessarily distinct) $1-3$ chains going out of it.

Step 4: Call MAXIMAL_SET$(3,1,3)$. Call RESOLVE.
REMARK: Impasse is resolved for a vertex, if colour was changed in any one of the two $1-3$ components emanating out of it. For each $v \in I$, now we have a simple $1-3$ path in G with v_1 and v_3 as its end points. But, note that now it is not necessary for v_1 and v_2 to be in the same $1-2$ component, let alone a $1-2$ path.

Step 5: For each impasse vertex v, associate all Q-flags in the $1-3$ path from v_1 to v_3 with v, and put up all of them.

Step 6: Recolour all colour-3 vertices with colour 1 where feasible. Recolour all colour-2 vertices with colour 1 where feasible. Call RESOLVE. Call MAXIMAL_SET$(3,3,2)$. Call RESOLVE.
REMARK: For each $v \in I$, now we have a simple $2-3$ path in G with v_2 and v_3 as its end points. But, v_1 and v_2 may not be in the same $1-2$ component. And v_1 and v_3 may not be in the same $1-3$ component. All P-components and Q-components of G are simple chains, by construction. Also, no vertex can have both its flags up, unless it is a common endpoint of a P-chain and a Q-chain. So, every impasse vertex v has a P-path $L_P(v)$ and a Q-path $L_Q(v)$ of its own.

Step 7: For each impasse vertex v, if $L_P(v)$ has at least one vertex with a non-P-neighbour of colour 1 or 2, find either a 1-fork or a 2-fork at v that involves only vertices from $L_P(v)$. Use this fork to resolve the impasse at v. Call RESOLVE.
REMARK: When $L_P(v)$ has at least one vertex with a non-P-neighbour of colour 1 or 2, the two endpoints of $L_P(v)$ being coloured 1 and 2, a fork at v can be found from $L_P(v)$ in at least one direction. If v is still at impasse after this step, $L_P(v)$ is a maximal $1-2$ component of G.

Step 8: For each impasse vertex v, if $Q_P(v)$ has at least one vertex with a non-Q-neighbour of colour 1 or 3, find either a 1-fork or a 3-fork at v that involves only vertices from $L_Q(v)$. Use this fork to resolve the impasse at v.
REMARK: So, we are left with only the case where v_i and v_j are the end points of a simple i-j path for $1 \leq i < j \leq 3$.

Step 9: For $v \in I$, let $v_1 = x$ $v_2 = y$ and $v_3 = z$. If y is adjacent to both x and z, then x and z are not adjacent because G has no 4-cliques, and y is the only neighbour of colour 2 for both x and z; let $\sigma(x) = \sigma(z) = 2$, $\sigma(y) := 1$, and $\sigma(v) := 3$. If y is adjacent to at most one of x and z, then interchange colours 1 and 3 in $L_Q(v)$. Now, clearly, $L_P(v)$ can provide a fork at v in at least one direction. Use this fork to resolve the impasse at v. Call RESOLVE.

Theorem 5.1 *The procedure 3-Colour-Cubic-Graphs 3-colours a 4-clique free cubic graph in $O(\log n)$ time and $O(n)$ operations on an EREW PRAM.*

Proof: Correctness of the procedure is obvious from the remarks following individual steps.

In Step 1 of the procedure, we have to find an MIS of G. We do this by first finding a $(\Delta+1)$-colouring $C : V \rightarrow \{1, \ldots, \Delta+1\}$ of G using the optimal algorithm [87] for $(\Delta+1)$-colouring a bounded degree graph, and then, for $i := 1$ to $\Delta+1$, in turn adding $w \in V$ to M, if $C(w) = i$ and no neighbour of w is already in M. The time taken is $O(\Delta^2 \log \Delta \log^{(k)} n)$ (for any fixed $k \geq 1$) with $\frac{n}{\log^{(k)} n}$ processors on an EREW PRAM. That is, with $\frac{n}{\log n}$ processors ($k = 1$), the time taken is $O(\Delta^2 \log \Delta \log n)$. Besides, there are $O(1)$ instances of, all vertices having to scan their respective neighbourhoods in parallel; these can be solved in $O(\Delta)$ time and $O(n\Delta)$ operations on a CREW PRAM. The rest of the procedure is dominated by a constant number of invocations to the list ranking algorithm, and hence can be solved optimally in $O(\log n)$ time on an EREW PRAM [6]. Hence the claim on resource requirements holds for a CREW PRAM.

For two adjacent vertices u and w, let $[u, w]$ be the entry for the edge (u, w) in u's edge list. We assume that $[u, w]$ and $[w, u]$ have a pointer to each other. These pointers, if not given as a part of the input, can be easily created in $O(1)$ time using $O(n^2)$ space and n processors on an EREW PRAM—form an adjacency matrix in

which each "non-zero" item is a pointer to an edge list entry; no initialisation is needed as we will never look at a "zero". It is easy to see that with this structure available, all vertices can scan their neighbourhoods in parallel, in $O(\Delta)$ time, without read conflicts. Hence, the algorithm can be run on an EREW PRAM also, with the same resource bounds.

Note that Procedure RESOLVE can be implemented on an EREW PRAM in $O(\log n)$ time. Since, the P-vertices and the Q-vertices associated with an impasse vertex form a list each, an invocation to list ranking is sufficient to bring down the flags in all of them. □

Since for any subcubic graph on n vertices, a cubic graph on $O(n)$ vertices of which the former is a subgraph, can be constructed in constant time with $O(n)$ processors [58]. Thus, we have the following corollary:

Corollary 5.1 *A 4-clique free 3-degree graph can be 3-coloured in $O(\log n)$ time and $O(n)$ operations on an EREW PRAM.*

Now, an algorithm for Brooks' colouring a general graph is presented.

Procedure Δ-Colour-Regular-Graphs

Input: A $(\Delta+1)$-clique-free regular graph $G = (V, E)$ in adjacency list representation.

Output: A Δ-colouring $\sigma : V \rightarrow \{1, \ldots, \Delta\}$ of G.

Step 0: If $\Delta \leq 3$ use the procedure 3-Colour-Cubic-Graphs.

Step 1: Find an MIS M of G, and for each $v \in M$ let $\sigma(v) = \Delta$. The graph $G - M$ has a maximum degree of $(\Delta-1)$. From every Δ-clique of $G - M$ elect a representative into a set I. As $G - M - I$ does not contain any Δ-clique, it is $(\Delta - 1)$-colourable. Recursively $(\Delta - 1)$-colour $G - M - I$. Each vertex of I is at impasse, and is left uncoloured. For each $v \in I$, select the minimum $\beta \in \{1, \ldots, \Delta - 1\}$ such that v_Δ and v_β are not adjacent. Swap colours between v_1 and v_β.

REMARK: For each $v \in I$, $G[v, v_1, ..., v_{\Delta-1}]$ is a Δ-clique of G. Also, v_1 and v_Δ are not adjacent. Observe that, v_Δ is not adjacent to all of $\{v_1, \ldots, v_{\Delta-1}\}$ because, v_Δ is adjacent to v and G does not contain a $(\Delta+1)$-clique.

Step 2: Recolour, first all colour-1 vertices, and then all colour-Δ vertices, with a colour other than 1 and Δ where feasible. Call RESOLVE.

REMARK: Now, every $1-\Delta$ component of G is a chain or a cycle. Also, for each $v \in I$, v_1 and v_Δ are end points of $1-\Delta$ chains.

Step 3: Call MAXIMAL_SET($\Delta, 1, \Delta$). Call RESOLVE.

Step 4: For each impasse vertex v, associate all P-flags in the $1-\Delta$ path from v_1 to v_Δ with v, and put up all of them.

Step 5: Repeat steps 2 and 3, with colour 2 substituted for colour 1.
REMARK: Now v_2 and v_Δ are the end points of the same 2-Δ path, for every $v \in I$. But v_1 and v_Δ need not even be in the same $1-\Delta$ component. All P-components of G are simple chains, by construction. So, every impasse vertex v has an exclusive P-path $L_P(v)$.

Step 6: For each impasse vertex v, if possible, find a fork at v that involves only vertices from $L_P(v)$, and use this fork to diffuse the impasse at v. Call RESOLVE.
REMARK: If $L_P(v)$ does not provide a fork as required, then no vertex w of $L_P(v)$ has two non-P-neighbours of the same colour. Also, both P-neighbours of w, when there are two, must be coloured the same. Since v_1 and v_Δ are the end points of $L_P(v)$, this would mean that $L_P(v)$ is a maximal $1-\Delta$ component of G. That is, v_1 and v_Δ are the end points of the same $1-\Delta$ path $S_1(v) = L_P(v)$, and v_2 and v_Δ are the end points of the same $2-\Delta$ path $S_2(v)$. The only vertex common to $S_1(v)$ and $S_2(v)$ is v_Δ. Since v_1 and v_Δ are not adjacent, $S_1(v)$ does not degenerate into a single edge. But, $S_2(v)$ may be a single edge.

Step 7: For each impasse vertex v, interchange colours 2 and Δ in $S_2(v)$. No neighbour of v_1 is now of colour 2, because, prior to this step, v_1 and v_Δ were not adjacent. Recolour v_1 with 2, and the impasse at v is resolved.

Theorem 5.2 *The procedure Δ-Colour-Regular-Graphs Δ-colours a regular Brooks' graph in $O(\Delta^2 \log \Delta \log n)$ time with $\frac{n}{\log n}$ processors on an EREW PRAM.*

Proof: From the proof of Theorem 5.1 it follows that with $\frac{n}{\log n}$ processors on an EREW PRAM, all steps of Procedure Δ-Colour-Regular-Graphs, except the $(\Delta+1)$-colouring and the recursive call in Step 1, can be implemented in $O(\Delta \log n)$ time, and that the $(\Delta+1)$-colouring can be done in $O(\Delta^2 \log \Delta \log n)$ time. Note that $G - M$ is properly

coloured and does not have a vertex of colour $\Delta + 1$. That is, the restriction of the colouring C to $V - M$ is a valid Δ-colouring of $G - M$. Hence, $G - M - I$ is already Δ-coloured when it is passed on to the next lower level of recursion and need not be coloured again. In other words, we need only one invocation of the colouring algorithm, at the beginning of the procedure; the lower levels of recursion can make use of the same colouring.

Hence the theorem. □

Corollary 5.2 *A Brooks' graph can be Δ-coloured in $O(\Delta^2 \log \Delta \log n)$ time with $\frac{n}{\log n}$ processors on an EREW PRAM.*

5.2 Local Nature of Brooks' Colouring: A Combinatorial result

We say that a pair of vertices u and v in a k-vertex-colourable graph G are *k-friends* iff they have the same colour in every k-colouring of G, and *k-foes* iff they have different colours in every k-colouring of G. For example, for a vertex v in a chain L, every vertex at an even distance from v in L is a 2-friend of v, and every vertex at an odd distance from v in L is a 2-foe of v; in contrast, v does not have a 3-friend, and its neighbours are its only 3-foes.

In the previous section, we showed that the problem of 3-colouring a 3-degree Brooks' graph can be reduced to a constant number of invocations to list ranking. Hence it is intermediate in complexity to 2-colouring and 3-colouring of chains.

The problem of 3-colouring a chain has an $O(\log(\log^* n))$ time linear processor EREW PRAM algorithm [48]. In contrast, the problem of 2-colouring a chain has a lower bound of $\Omega(\log n / \log\log n)$ on time, with a polynomial number of processors, on a CRCW PRAM [32]. This lower bound is proved using a reduction from Parity. Given an instance of Parity, its 1's are linked up in the same order in which they appear. The resulting list is then two-coloured. Comparing the colours of the first and last vertices, we get the solution of the instance of Parity. This reduction relies on the fact that, in a chain, vertices can have 2-friends and 2-foes arbitrarily far away; fixing the colour of one vertex fixes the colour of all others.

Investigating the distribution of 3-friends and 3-foes in a cubic Brooks' graph, we prove the following two theorems. The proofs are presented in Section 5.2.1

Theorem 5.3 *For every 4-clique-free 3-degree graph G, and for any two vertices u and v in G apart by more than 361 edges, there exists a 3-colouring of G that has u and v coloured the same.*

Theorem 5.4 *For every 4-clique-free 3-degree graph G, and for any two vertices u and v in G apart by more than 362 edges, there exists a 3-colouring of G that has u and v coloured differently.*

Thus, in a 3-degree Brooks' graph, vertices can have their 3-friends and 3-foes at most a constant distance away. Hence, it appears that there may be no $o(\log n/\log\log n)$ time reduction from Parity to 3-colouring of 3-degree Brooks' graphs. Thus, probably, for 3-colouring of 3-degree Brooks' graphs, we can expect to find an algorithm that is asymptotically faster than the $O(\log n)$ time algorithm of the previous section, unlike the 2-colouring of chains, where the $O(\log n)$ time optimal EREW PRAM algorithm of [6] is believed to be the best possible.

Panconesi and Srinivasan [80] show that for a Brooks' graph G and a vertex v in it, a Δ-colouring of $G - v$ can be extended to the whole of G by recolouring a path of length $O(\log_\Delta n)$. They also prove that this bound is tight in that there exists a graphs G and a vertex v in it so that extending some Δ-colourings of $G - v$ to the whole of G will require recolouring a path of length $\Omega(\log_\Delta n)$. Our theorems can be seen as complementing these results. Note that our conjecture is not weakened in any way by these results, because a Brooks' colouring algorithm, on the one hand, need not rely on a partial initial colouring that is subsequently modified to cover the whole of the graph, and on the other, even when it does may avoid the degenerate case of [80].

Thus, the 3-friends and 3-foes of a vertex in a cubic Brooks' graph cannot be more than a constant distance away.

5.2.1 Proofs of the Theorems

Theorem 5.3 is proved first; Theorem 5.4 will then be derived as a corollary.

Let $G = (V, E)$ be a 3-degree Brooks' graph on n vertices, and let u and v be two of its vertices apart by more than 361 edges. Consider a 3-colouring σ of G such that $\sigma(v) \neq \sigma(u)$. We show that the colouring σ can be modified into another valid colouring of G that has both u and v coloured the same. Let x, y and z be the three neighbours of v.

We use the following proposition, which will be proved in Section 3, in our proof of Theorem 5.3:

PROPOSITION $\mathcal{P}(u, v, x, y, z)$
If x and u are connected in $G - \{v, y, z\}$, then the 3-colouring σ of G can be modified into a 3-colouring σ' of $G - v$ so that

- $\sigma(x) \neq \sigma'(x)$
- $\sigma(y) = \sigma'(y)$, $\sigma(z) = \sigma'(z)$, $\sigma(u) = \sigma'(u)$

Continuing with the proof of Theorem 5.3, we assume that u and v are connected in G, because otherwise, a permutation of colours in the component of u (or of v) is enough to give the same colour to both u and v. We consider the following three exclusive possibilities:

1. Among the three neighbours of v, there is a cut-vertex of G (say z), the removal of which disconnects the other two from u. That is, x and y are disconnected from u in $G - z$; in other words, every path from x or y to u in G contains z.

2. Among the three neighbours of v, there is a pair (say, y and z), the removal of which disconnects the third from u. That is, x is disconnected from u in $G - \{y, z\}$; in other words, every path from x to u in G contains y or z.

3. For each of the three neighbours of v, there is a path in G that connects it to u, but does not contain the other two. That is, x and u are connected in $G - \{y, z\}$, and y and u are connected in $G - \{x, z\}$, and z and u are connected in $G - \{x, y\}$.

Without loss of generality assume that $\sigma(u) = 1$ and $\sigma(v) = 2$.

Case C1: x and y are disconnected from u in $G - z$

Let us denote the component that contains u in $G - z$ by A.

Since every path from x or y to u in G contains z, there must be a path from z to u that does not contain x, y or v. Hence, z and u are connected in $G - \{v, x, y\}$.

Since v is coloured 2, x, y and z are coloured 1 or 3. If z is coloured 1, then use Proposition $\mathcal{P}(u, v, z, x, y)$ to recolour z without affecting x, y or u; otherwise, continue. Uncolour v.

The colouring now satisfies the following conditions: v is uncoloured; u is coloured 1; x is coloured 1 or 3; y is coloured 1 or 3; z is coloured 2 or 3.
Since v is uncoloured, one of the following four exclusive possibilities must hold.

(I). **Colour 1 is free at** v. Here, v can be coloured 1, the same as u.
(II). **Colour 2 is free at** v. Here, z must be coloured 3. (Either z was initially coloured 1 and an invocation to Proposition $\mathcal{P}(u,v,z,x,y)$ has changed its colour to the present 3, or z was initially coloured 3 and no colour has been changed since then.) Swap colours 1 and 2 in component A; u gets colour 2, but x, y and z retain their colours. Now v can be coloured 2, the same as u.
(III). **Colour 3 is free at** v. Here, z must be coloured 2, and both x and y coloured 1. Swap colours 1 and 3 in component A; u gets colour 3, but x, y and z retain their colours. Now v can be coloured 3, the same as u.
(IV). **No colour is free at** v. Since x and y are not coloured 2, z must be coloured 2. Without loss of generality, assume that x and y are coloured 1 and 3 respectively.

First assume that x and z are not in the same 1-2 component. Swap colours 1 and 2 in the 1-2 component that contains x; x gets recoloured 2, while y, z and u retain their colours. Now, v can be coloured 1, the same as u.

Next assume that x and z are in the same 1-2 component. So, if at all z has a neighbour of colour 3, it must be in A; the other two valencies of z are taken up by v and a vertex coloured 1. Since y is not in A, this means that y and z are not in the same 2-3 component. Swap colours 2 and 3 in the 2-3 component that contains z; z gets recoloured 3, while x, y and u retain their colours. Now, colour 2 is free at v, and the situation is similar to (II) above. So, swap colours 1 and 2 in component A; u gets colour 2, but x, y and z retain their colours. Now v can be coloured 2, the same as u.

Case C2: x is disconnected from u in $G - \{y,z\}$

Let us denote the component that contains u in $G - \{y,z\}$ by B.

Since every path from x to u in G contains y or z, and neither y nor z is a cut-vertex of G (otherwise, the situation falls under Case C1), there must be a path from z to u that does not contain x, y or v, and a path from y to u that does not contain z, x or v. Hence, z and u are connected in $G - \{v,x,y\}$. Also, y and u are connected in $G - \{v,z,x\}$.

Since v is coloured 2, x, y and z are coloured 1 or 3. If z is coloured 1, then use Proposition $\mathcal{P}(u,v,z,x,y)$ to recolour z without affecting x, y or u; otherwise, continue. If y is coloured 1, then use Proposition $\mathcal{P}(u,v,y,z,x)$ to recolour y without affecting z, x or u; otherwise, continue. Uncolour v.

The colouring now satisfies the following conditions: v is uncoloured; u is

coloured 1; x is coloured 1 or 3; y is coloured 2 or 3; z is coloured 2 or 3.
Since v is uncoloured, one of the following four exclusive possibilities must hold.
(I). **Colour 1 is free at v.** Here, v can be coloured 1, the same as u.
(II). **Colour 2 is free at v.** Here, both y and z must be coloured 3. Swap colours 1 and 2 in component B; u gets colour 2, but x, y and z retain their colours. Now v can be coloured 2, the same as u.
(III). **Colour 3 is free at v.** Here, both y and z must be coloured 2. Swap colours 1 and 3 in component B; u gets colour 3, but x, y and z retain their colours. Now v can be coloured 3, the same as u.
(IV). **No colour is free at v.** Since neither y nor z is coloured 1, x must be coloured 1. Without loss of generality, assume that y and z are coloured 2 and 3 respectively.

First assume that x and z are not in the same 1-3 component. Swap colours 1 and 3 in the 1-3 component that contains x; x gets recoloured 3, while y, z and u retain their colours. Now, v can be coloured 1, the same as u.

Next assume that x and y are not in the same 1-2 component. Swap colours 1 and 2 in the 1-2 component that contains x; x gets recoloured 2, while y, z and u retain their colours. Now, v can be coloured 1, the same as u.

Next assume that y and z are not in the same 2-3 component. Swap colours 2 and 3 in the 2-3 component that contains y; y gets recoloured 3, while x, z and u retain their colours. Now, colour 2 is free at v, and the situation is similar to (II) above. So, swap colours 1 and 2 in component B; u gets colour 2, but x, y and z retain their colours. Now v can be coloured 2, the same as u.

Now the only case remaining is where x and y are in the same 1-2 component C_{12}, and x and z are in the same 1-3 component C_{13}, and y and z are in the same 2-3 component C_{23}. Here C_{12} must be a chain with x and y as its endpoints; if C_{12} has vertex w with only one colour in the neighbourhood, then w can be given colour 3, thus ensuring that x and y are in different 1-2 components. Similarly, C_{13} must be a chain with x and z as its endpoints, and C_{23} must be a chain with y and z as its endpoints. Also, C_{23} does not degenerate into a single edge, because otherwise, u and v would be disconnected in G. Note that $C_{23} - \{y, z\}$ is a subgraph of component B.

Swap colours 1 and 3 in C_{13}; x and z are now coloured 3 and 1 respectively. Since $C_{23} - \{y, z\}$ is a subgraph of component B, all its vertices retain their colour. Now, the only neighbour coloured 2 of z is in $C_{23} - \{y, z\}$, and hence in B; the other

two valencies of z are taken up by v and a vertex coloured 3. Thus, y and z are in different 1-2 components.

Swap colours 1 and 2 in $C_{12} - x$, the 1-2 component that contains y; y is now coloured 1. Now, x has two neighbours of colour 1, one each from C_{12} and C_{13}.

Let a, b and c be the three neighbours of u, in G. Since, u is coloured 1, all its neighbours must be coloured either 2 or 3. We consider the four exclusive possibilities one by one:

(i). a, b and c are all coloured 3. Colour both u and v with 2.

(ii). a, b and c are all coloured 2. Recolour x with 2; now colour 3 is free at v. Colour both u and v with 3.

(iii). Two of a, b and c are coloured 3. Without loss of generality, assume that a, b and c are coloured 2, 3 and 3 respectively. Let v be coloured 2. Then we can assume that u and v are in the same the 1-2 component, because otherwise, swapping colours 1 and 2 in one of the components will give the same colour to u and v. Since u's only neighbour of colour 2 is a, the 1-2 component that contains u and v must also contain a. Thus, by Proposition $\mathcal{P}(v, u, a, b, c)$, a can be recoloured without also recolouring b, c or v. Irrespective of whether the new colour a is 1 or 3, u can be coloured 2, the same as v.

(iv). Two of a, b and c are coloured 2. Without loss of generality, assume that a, b and c are coloured 2, 2 and 3 respectively. Recolour x with 2; now colour 3 is free at v. Let v be coloured 3. Then we can assume that u and v are in the same the 1-3 component, because otherwise, swapping colours 1 and 3 in one of the components will give the same colour to u and v. Since u's only neighbour of colour 3 is c, the 1-3 component that contains u and v must also contain c. Thus, by Proposition $\mathcal{P}(v, u, c, a, b)$, c can be recoloured without also recolouring a, b or v. Irrespective of whether the new colour c is 1 or 2, u can be coloured 3, the same as v.

Case C3: x and u are connected in $G - \{y, z\}$, and
y and u are connected in $G - \{x, z\}$, and
z and u are connected in $G - \{x, y\}$

If $\sigma(x) = 1$, then we use Proposition $\mathcal{P}(u, v, x, y, z)$ to recolour x without affecting y, z or u. If $\sigma(y) = 1$, then we use Proposition $\mathcal{P}(u, v, y, z, x)$ to recolour y without affecting x, z or u. If $\sigma(z) = 1$, then we use Proposition $\mathcal{P}(u, v, z, x, y)$ to recolour y without affecting x, y or u. None of x, y and z are now coloured 1. Thus, v

can be coloured 1, the same as u.

Thus, given that Proposition $\mathcal{P}(u,v,x,y,z)$ is valid, we have Theorem 5.3, The proof Proposition $\mathcal{P}(u,v,x,y,z)$ is given in Section 5.2.2

To prove Theorem 5.4, let $G = (V,E)$ be a 3-degree Brooks' graph on n vertices, and let u and v be two of its vertices apart by more than 362 edges. Let x be a neighbour of v. Then u and x are apart by more than 361 edges. By Theorem 5.3, there is a 3-colouring of G that colours x and u the same; u and v are coloured differently in this colouring. Hence Theorem 5.4.

5.2.2 Proof of the Proposition

We now prove Proposition $\mathcal{P}(u,v,x,y,z)$

Suppose x and u are connected in $G-\{v,y,z\}$. Consider the component that contains x in $G-\{v,y,z,u\}$. Of all breadth first search (BFS) trees rooted at x, of this component, **pick one that has the largest number of leaves.** Note that there could be arbitrarily large graphs H such that for a given vertex r in H, all BFS trees of H rooted at r have a constant number of leaves (see Figure 5.1). Let $T = (V_T, E_T)$

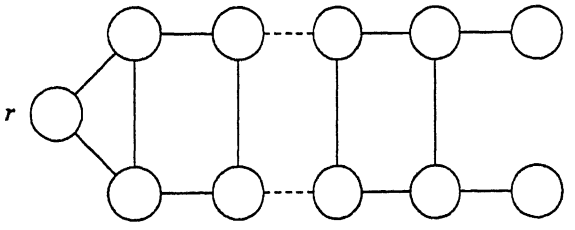


Figure 5.1: All BFS trees at r have a constant number of leaves

be the BFS-tree thus obtained. This means that if $\{y,z,u\}$ is a cutset in $G-v$, we have left out from T precisely those vertices in $G-v$ reachable from x only through y, z or u. Note that T contains at least one neighbour of u, and hence contains at least 361 vertices.

For a vertex $s \in V_T$, let $p(s)$ denote the parent of s in T. Also, for $s,t \in V_T$, let $P(s,t)$ be the path in T from s to t and $d(s,t)$ be the shortest distance in $G-\{v,y,z,u\}$ from s to t. Since T is a BFS-tree, for all $s \in V_T$, the length of $P(x,s)$ is $d(x,s)$.

If for a vertex $s \in V_T$, $P(x,s)$ is a fork in G, then this fork can be used to recolour x without affecting y, z or u. Hence, we can make the following assumption:

Assumption 1: For each $s \in V_T$, $P(x,s)$ is not a fork in G.

In other words, if s_1, s_2 and s_3 are the neighbours of s, $s_1 = p(s)$, then $\sigma(s_2) \neq \sigma(s_3)$.

Let l be a node of T that is at least three edges away from x, y, z and u in $G - v$. Since the combined valency of x, y, z and u in $G - v$ is only nine, there are at most nine vertices one edge away, and at most 18 vertices two edges away from x, y, z or u. Thus, if T has more than 28 vertices, a vertex l as required can always be found. Let $e = p(l)$, f and g be the neighbours of l in G. Note that e, f and g are present in T, because all are reachable from x without going through y. z or u. Since $P(x,l)$ is not a fork, one (and only one) of f and g, say f, is coloured the same as e. Without loss of generality, assume that $\sigma(e) = \sigma(f) = 1$, $\sigma(l) = 2$ and $\sigma(g) = 3$.

We show that for certain appropriate choices of l, one of which will always be possible, we can find a fork at x that consists only of a path $P(x,w)$ in T, for some $w \in V_T$, and some vertices at most two edges away from l. Since l is at least three edges away from y, z and u in $G - v$, such a fork clearly will not contain y, z or u, and hence can be used to recolour x independently of them.

We make the following claims that will be subsequently used in the proof.

Claim 1: For $w_1, w_2 \in V_T$, if $w_1 \in P(x,w_2)$ then the path from w_1 to w_2 in T is one of the shortest paths from w_1 to w_2 in $G - \{v,y,z,u\}$.

Proof of Claim 1: T is a BFS-tree.

Claim 2: For $w_1, w_2 \in V_T$, if $w_1 \in P(x,w_2)$ and $\{w_1,w_2\} \in E$, then $w_1 = p(w_2)$.

Claim 3: For $w_1, w_2 \in V_T$, $w_1 \neq w_2$, if none w_1's children is in $P(x,w_2)$, then w_1 is in not $P(x,w_2)$.

Continuing with the proof of the theorem, **if possible, let l be a leaf.** For this particular case we make the following claims:

Claim 4: $l \notin P(x,f)$ and $l \notin P(x,g)$

Proof of Claim 4: Since l is a leaf in T, it is not an ancestor of f or g.

Claim 5: If $\{e,g\} \notin E$ then $e \notin P(x,g)$

Proof of Claim 5: Suppose $\{e,g\} \notin E$ and $e \in P(x,g)$. Then we have the following sequence of statements (see Figure 5.2(a)):

(i) $d(e,g) \leq 2$: l is adjacent to e and g in G

(ii) $d(e,g) \geq 2$: $\{e,g\} \notin E$

(iii) $d(e,g) = 2$: from (i) and (ii)

(iv) there exists a vertex $h \neq l$ in T

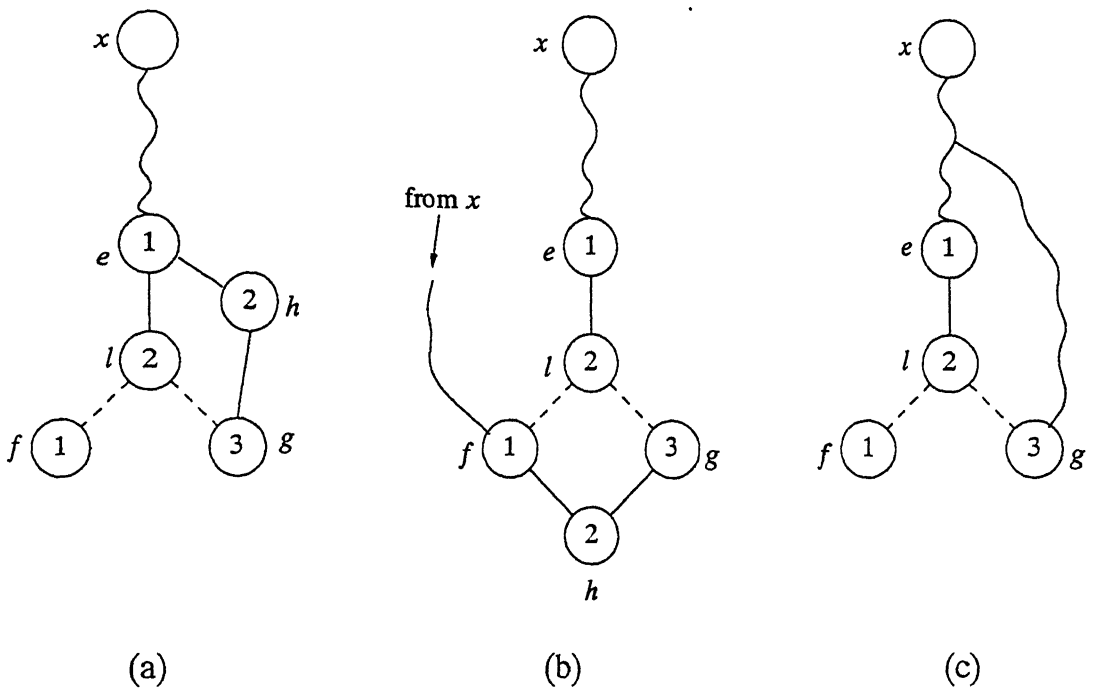


Figure 5.2: (a) Claim 5: $P(x,e)$ is a fork (b) Claim 6: $P(x,f)$ is a fork (c) Claim 7: $\langle P(x,g),l\rangle$ is a fork

	so that $p(g)=h$ and $p(h)=e$	: $e \in P(x,g)$, from (iii) and Claim 1;
		: l is a leaf
(v)	$p(h)=p(l)=e$	: from (iv)
(vi)	$\sigma(h)=\sigma(l)=2$	: $\sigma(e)=1$ and $\sigma(g)=3$
(vii)	$P(x,e)$ is a fork; a contradiction	: from (v), (vi) and Assumption 1

Claim 6: If $\{f,g\} \notin E$ then $f \notin P(x,g)$

Proof of Claim 6: Suppose $\{f,g\} \notin E$ and $f \in P(x,g)$. Then we have the following sequence of statements (see Figure 5.2(b)):

(i)	$d(f,g) \leq 2$	: l is adjacent to f and g in G
(ii)	$d(f,g) \geq 2$	: $\{f,g\} \notin E$
(iii)	$d(f,g)=2$	: from (i) and (ii)
(iv)	there exists a vertex $h \neq l$ in T	
	so that $p(g)=h$ and $p(h)=f$	: $f \in P(x,g)$, from (iii) and Claim 1
		: l is a leaf
(v)	$h \notin P(x,f)$ and $l \notin P(x,f)$	: from (iv), Claim 4

(vi) h and l are adjacent to f in G : by definition

(vii) $\sigma(h) = \sigma(l) = 2$: $\sigma(f) = 1$ and $\sigma(g) = 3$

(ix) $P(x,f)$ is a fork; a contradiction : from (v), (vi), (vii) and Assumption 1

In the sequel, for r distinct vertices $w_1, \ldots, w_r$ none of which is in $P(x,w)$, by $\langle P(x,w), w_1, \ldots, w_r \rangle$ we mean the vertex disjoint path obtained by concatenating $P(x,w)$ with edges $\{w_1,w_2\}, \ldots, \{w_{r-1}, w_r\}$, assuming that all the required edges exist in G.

Claim 7: If none of l, e and f is in $P(x,g)$ then $\langle P(x,g), l \rangle$ is a fork.
Proof of Claim 7: See Figure 5.2(c).

(i) e and f are adjacent to l in E : by definition

(ii) e and f are not in $\langle P(x,g), l \rangle$: by the hypothesis of the claim

(iii) $\sigma(e) = \sigma(f) = 1$: from the choice of σ

(iv) $\langle P(x,g), l \rangle$ is a fork : from (i), (ii) and (iii)

Continuing with the proof of the theorem, we consider the following four exclusive possibilities, one by one. For each case we find a fork that can be used to recolour x without affecting y, z or u.

Case A1: $\{e,g\} \notin E$ and $\{f,g\} \notin E$

By Claim 4, $l \notin P(x,g)$. Since $\{e,g\} \notin E$, by Claim 5, $e \notin P(x,g)$. Since $\{f,g\} \notin E$, by Claim 6, $f \notin P(x,g)$. Hence, by Claim 7, $\langle P(x,g), l \rangle$ is a fork.

Case A2: $\{e,g\} \notin E$ and $\{f,g\} \in E$

By Claim 4, $l \notin P(x,g)$. Since $\{e,g\} \notin E$, by Claim 5, $e \notin P(x,g)$. Hence, if $f \notin P(x,g)$, then, by Claim 7, $\langle P(x,g), l \rangle$ is a fork.

Assume that $f \in P(x,g)$. Since $\{f,g\} \in E$, by Claim 2, f is the parent of g in T. Let h be a neighbour of g, different from both f and l. Since g is coloured 3, h must be coloured either 1 or 2. (see Figure 5.3(a).) We have the following sequence of statements:

(i) $l \notin P(x,g)$: Claim 4

(ii) $h \notin P(x,g)$: $\{h,g\}$ is in G, $p(g) = f$, Claim 2

(iii) h is adjacent to g in G : by definition

(iv) l is adjacent to g in G : by definition

(v) if $\sigma(h) = 2$ then

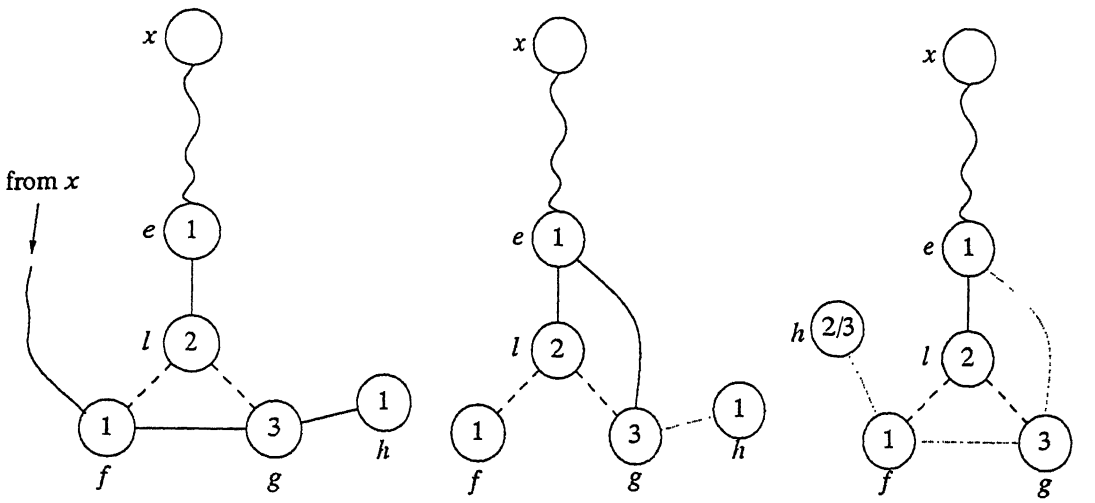


Figure 5.3: (a) Case A2, $f = p(g)$: $\langle P(x,l), g\rangle$ is a fork (b) Case A3, $e = p(g)$: $\langle P(x,f), l, g\rangle$ or $\langle P(x,h), g, l\rangle$ is a fork (c) Case A4: $\langle P(x,e), g, f\rangle$ or $\langle P(x,l), f\rangle$ is a fork

	$P(x,g)$ is a fork	: $\sigma(l) = 2$, (i)–(iv)
(vi)	$\sigma(h) = 1$	: (v) and Assumption 1
(vii)	$\sigma(e) = \sigma(f) = 1$	: the choice of σ
(viii)	f is adjacent to g in G	: the hypothesis of Case A2
(ix)	$f \notin P(x,l)$	: $\{f,l\} \in E$, $p(l) = e$, and Claim 2
(x)	$d(h,l) = 2$	: $\{h,l\} \notin E$ and (iii) and (iv)
(xi)	if $h \in P(x,l)$ then $h = p(p(l))$	: (x) and Claim 1
(xii)	if $h \in P(x,l)$ then $h = p(e)$	: (xi), $e = p(l)$
(xiii)	$h \notin P(x,l)$	: (xii); from (vi) and (vii), $\sigma(h) = \sigma(e)$
(xiv)	$g \notin P(x,l)$	: $\{g,l\} \in E$, $p(l) = e$, and Claim 2
(xv)	f and h are not in $\langle P(x,l), g\rangle$	: from (ix) and (xiii)
(xvi)	$\langle P(x,l), g\rangle$ is a fork	: (iii), (vi)–(viii), (xiv) and (xv)

Case A3: $\{e,g\} \in E$ and $\{f,g\} \notin E$

By Claim 4, $l \notin P(x,g)$. Since $\{f,g\} \notin E$, by Claim 6, $f \notin P(x,g)$. Hence, if $e \notin P(x,g)$, then, by Claim 7, $\langle P(x,g), l\rangle$ is a fork.

Assume that $e \in P(x,g)$. Since $\{e,g\} \in E$, by Claim 2, e is the parent of g in

T. (See Figure 5.3(b)). Let h be a neighbour of g, different from both e and l. Since g is coloured 3, h must be coloured either 1 or 2. We have the following sequence of statements:

(i)	$l \notin P(x,g)$	: Claim 4
(ii)	$h \notin P(x,g)$	: $\{h,g\}$ is in G, $p(g) = e$, Claim 2
(iii)	h is adjacent to g in G	: by definition
(iv)	l is adjacent to g in G	: by definition
(v)	if $\sigma(h) = 2$ then	
	$P(x,g)$ is a fork	: $\sigma(l) = 2$, (i)–(iv)
(vi)	$\sigma(h) = 1$	: (v) and Assumption 1
(vii)	$\sigma(e) = \sigma(f) = 1$	: by the choice of σ
(viii)	$l \notin P(x,f)$	: Claim 4

Now there are two cases: (a) $h \notin P(x,f)$ and (b) $h \in P(x,f)$.
Consider the case of $h \notin P(x,f)$; then the following statements are true.

(ix)	$g \notin P(x,f)$	: h is the only (if there is any) child of g,
		: $h \notin P(x,f)$, Claim 3
(x)	$e \notin P(x,f)$	: (viii), (ix) and Claim 3
(xi)	e is adjacent to g in G	: the hypothesis of Case A3

Therefore, from (iii),(vi)–(xi), we have that if $h \notin P(x,f)$ then $\langle P(x,f), l, g\rangle$ is a fork. On the other hand, when $h \in P(x,f)$, the following statements are true.

(xii)	$f \notin P(x,h)$	: $h \in P(x,f)$
(xiii)	$e \notin P(x,h)$	: otherwise, e is an ancestor of h, which
		: in turn is an ancestor of f; since
		: $d(e,f) = d(e,h) = 2$, this cannot be
(xiv)	e and f are adjacent to l in G	: by definition
(xv)	g and l are not in $P(x,h)$	: (xiii), $p(g) = p(l) = e$

Therefore, from (vii) and (xii)–(xv) we have that if $h \in P(x,f)$ then $\langle P(x,h), g, l\rangle$ is a fork.

Thus, we can find a fork irrespective of whether h is in $P(x,f)$ or not.

Case A4: $\{e,g\} \in E$ and $\{f,g\} \in E$

Let h be a neighbour of f, different from both g and l. Since f is coloured 1, h must be coloured either 2 or 3. (See Figure 5.3(c)). We have the following sequence of statements:

(i)	e, l and f are adjacent to g	: hypothesis
(ii)	g, h and l are adjacent to f	: hypothesis
(iii)	either $p(g) = f$ or $p(g) = e$	: (i); $p(g) \neq l$
(iv)	either $p(f) = h$ or $p(f) = g$	: (ii); $p(f) \neq l$

Now there are two cases: (a) $h \in P(x,e)$ and (b) $h \notin P(x,e)$.
Consider the case of $h \in P(x,e)$; then the following statements are true.

(v)	$e \notin P(x,h)$	: $h \in P(x,e)$
(vi)	$h \in P(x,l)$	: $h \in P(x,e)$; $e = p(l)$
(vii)	$d(h,l) = 2$	: f is adjacent to both h and l
(viii)	$h = p(p(l)) = p(e)$	: (vi), (vii), Claim 1
(ix)	if $p(f) = g$ then $p(p(p(f))) = h$	:
	else $p(f) = h$	: (iii), (viii), (iv)
(x)	$h \in P(x,f)$	: (ix)
(xi)	$f \notin P(x,h)$	: (x)
(xii)	$P(x,h)$ is a fork	: (v), (viii), (xi), $\sigma(e) = \sigma(f) = 1$

Hence, by Assumption 1, $h \notin P(x,e)$; and the following statements are true.

(xiii)	$h \notin P(x,e)$	:
(xiv)	$l \notin P(x,e)$	: $e = p(l)$
(xv)	if $p(g) = f$ then	:
	$p(p(g)) = h$, and so $g \notin P(x,e)$	: (iv), (xiii)
	else $p(g) = e$	: (iii)
(xvi)	$g \notin P(x,e)$	: (xv)
(xvii)	$f \notin P(x,e)$	: (ii), (xiii), (xiv) and (xvi)
(xviii)	if $\sigma(h) = 2$ then	:
	$\langle P(x,e), g, f \rangle$ is a fork	: (ii); (xiii), (xiv), (xvi), (xvii); $\sigma(l) = 2$
(xix)	$f, h, g \notin P(x,l)$	: (xiii), (xvi), (xvii); $e = p(l)$
(xx)	if $\sigma(h) = 3$ then	:

$\langle P(x,l), f\rangle$ is a fork : (ii), (xix); $\sigma(g) = 3$

Since h is coloured either 2 or 3, we get a fork any way.

Thus, if the vertex l, chosen to be at least three edges away from x, y, z and u in $G - v$, happens to be a leaf, then we can obtain a fork at x that does not involve y, z or u. So, assume that all leaves of T are at most two edges away from x, y, z or u in $G - v$. Since the combined valency of x, y, z and u in $G - v$ is only nine, there are at most nine vertices one edge away, and at most 18 vertices two edges away from x, y, z or u. Thus, there are at most 27 vertices less than three edges away from x, y, z or u. As can be readily verified, a maximum of 18 among these can be leaves in T. In any tree the number of vertices with two or more children cannot exceed the number of leaves. As T has at most 18 leaves, it has at most 18 vertices with exactly two children; note that T is a binary tree. In other words, all but at most 36 vertices in T have exactly one child.

Let l be a vertex such that all vertices at most two edges away from l have exactly one child in T. If T has more than $36(1 + 3 + 2 \times 3) = 360$ vertices, then we indeed have a choice for l. As mentioned earlier, T has at least 361 vertices.

Since l has exactly one child in T, either $p(f) = l$ or $p(g) = l$, but not both.

Case B1: f is the child of l

Here, $l \neq p(g)$. (See Figure 5.4(a)). We have the following sequence of statements:

(i) $l \notin P(x,g)$: $l \neq p(g)$, $\{l,g\} \in E$, Claim 2

(ii) $f \notin P(x,g)$: (i), $p(f) = l$

(iii) $e \notin P(x,g)$: (i), l is the only child of e, Claim 3

(iv) $\langle P(x,g), l\rangle$ is a fork : from (i)–(iii), Claim 7

Case B2: g is the child of l

Here $l \neq p(f)$. Let h and a be the neighbours of g, other than l, in G. Since $P(x,g)$ is not a fork, $\sigma(h) \neq \sigma(a)$. As g is coloured 3, without loss of generality, let $\sigma(h) = 1$ and $\sigma(a) = 2$. Since g is a neighbour of l in G, g has exactly one child in T.

First assume that a is the child of g. Then h is not a child of g. (See Figure 5.4(b)). We have the following statements:

(i) $g \notin P(x,h)$: $g \neq p(h)$, $\{g,h\} \in E$, Claim 2

(ii) $a \notin P(x,h)$: (i), $p(a) = g$

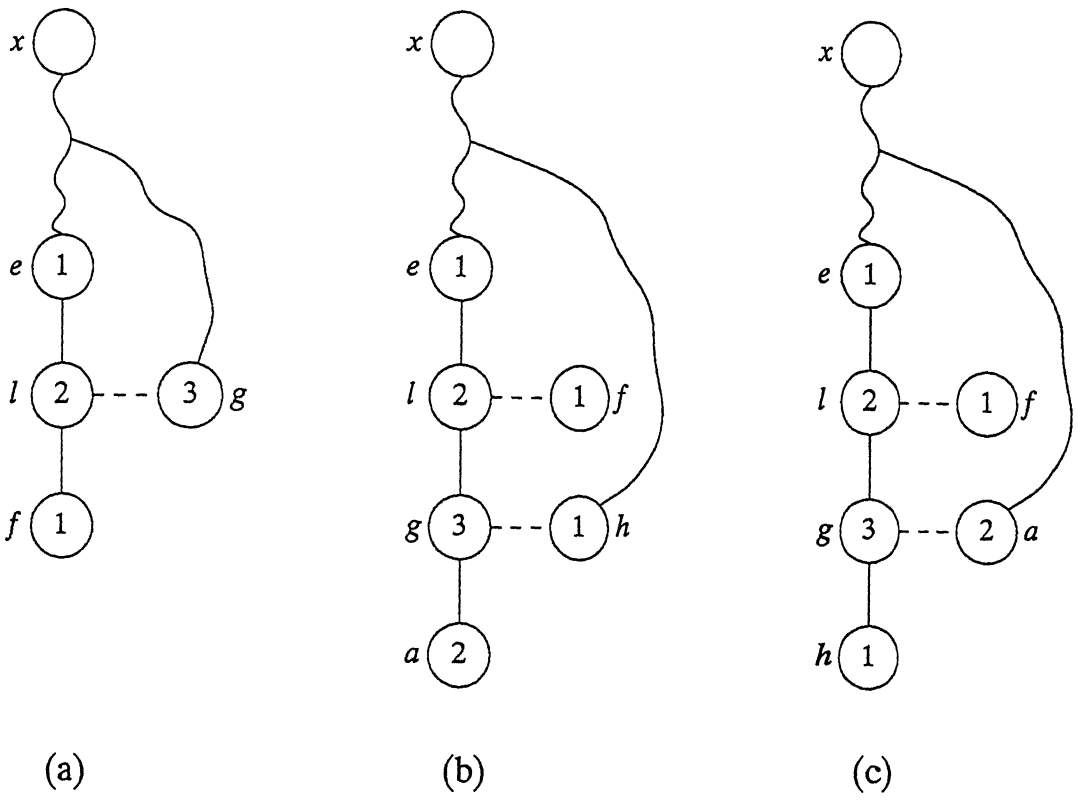


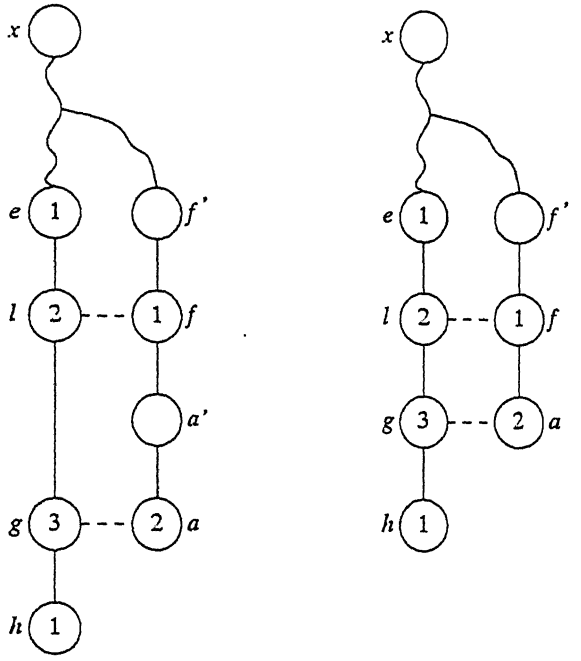
Figure 5.4: (a) Case B1: $\langle P(x,g),l\rangle$ is a fork (b) Case B2, a is the child of g: $\langle P(x,h),g\rangle$ is a fork (c) Case B2, h is the child of g, $f \notin P(x,a)$: $\langle P(x,a),g,l\rangle$ is a fork

(iii)	$l \notin P(x,h)$	: (i), g is the only child of l, Claim 3
(iv)	l and a are adjacent to g	: by definition
(v)	$\sigma(l) = \sigma(a) = 2$	:
(vi)	$\langle P(x,h),g\rangle$ is a fork	: from (i)–(v)

Now, consider the case where h is the child of g, and so, a is not a child of g.
If $f \notin P(x,a)$ then (see Figure 5.4(c)) we have the following sequence of statements:

(i)	$g \notin P(x,a)$	: $g \neq p(a)$, $\{g,a\} \in E$, Claim 2
(ii)	$l \notin P(x,a)$	: (i), g is the only child of l, Claim 3
(iii)	$e \notin P(x,a)$	: (ii), l is the only child of e, Claim 3
(iv)	e and f are adjacent to l	: by definition
(v)	$\sigma(e) = \sigma(f) = 1$	: by the choice of σ
(vi)	$\langle P(x,a),g,l\rangle$ is a fork	: from (i)–(v)

So, the only case remaining is where $f \in P(x,a)$ and h is the child of g.



(a) (b)

Figure 5.5: (a) Case B2, h is the child of g, $f = p(p(a))$ (b) Case B2, h is the child of g, $f = p(a)$: $P(x,f)$ is a fork

Since l is adajcent to f in $G[V_T]$, $|d(x,l) - d(x,f)| \leq 1$.

- Assume that $d(x,l) = d(x,f) - 1$. Let f' be the parent of f; so, $d(x,l) = d(x,f')$. Obtain a tree T_1 from T by redefining $p(f)$ as l. Note that T_1 is also a BFS-tree of $G[V_T]$. Since f' is two edges away from l in $G[v_T]$, f is its only child in T; that is, f' is a leaf in T_1. Thus, T_1 has one leaf more than T. But T, by definition, has the largest number of leaves of all BFS-trees rooted at x of $G[V_T]$. We have obtained a contradiction.

- Assume that $d(x,l) = d(x,f) + 1$. Since e be the parent of l, $d(x,e) = d(x,f)$. Obtain a tree T_2 from T by redefining $p(l)$ as f. T_2 is also a BFS-tree of $G[V_T]$, and has one leaf more than T. We derive a contradiction, once again.

Hence it must be that $d(x,l) = d(x,f)$.

Since $l = p(g)$ and $f \in P(x,a)$, $d(x,a) > d(x,f) = d(x,l) = d(x,g) - 1$. That is, $d(x,a) \geq d(x,f) + 1 = d(x,g)$. But g is adajcent to a in $G[V_T]$; hence

$d(x,a) \leq d(x,g) + 1 = d(x,f) + 2$. So, either f is the parent of a, or f is the grand parent of a, in T.

- Assume that $f = p(p(a))$; let a' be the parent of a. Clearly, $d(x,g) = d(x,a')$. Obtain a tree T_3 from T by redefining $p(a)$ as g. Since a' is two edges away from l in $G[v_T]$, a is its only child in T; that is, a' is a leaf in T_3. Thus, T_3 has one leaf more than T—a contradiction. (See Figure 5.5(a)).

Hence it must be that $f = p(a)$.

We have the following sequence of statements:

(i)	a is adjacent to f	: $f = p(a)$
(ii)	l is adjacent to f	: by definition
(iii)	l and a are not in $P(x,f)$	: $f = p(a)$; $d(x,l) = d(x,f)$
(iv)	$\sigma(l) = \sigma(a) = 2$	:
(v)	$P(x,f)$ is a fork	: from (i)–(iv)

(See Figure 5.5(b)). But, by Assumption 1, this is a contradiction.

That proves Proposition $\mathcal{P}(u,v,x,y,z)$.

Chapter 6

Colouring of Interval Graphs

A graph $G = (V, E)$ is called an interval graph, if for some set $\Im$ of intervals of a linearly ordered set, there is a bijection $f : V \rightarrow \Im$ so that two vertices u and v are adjacent in G iff $f(u)$ and $f(v)$ overlap. Every interval graph has an interval representation in which the endpoints are all distinct [36]. Hence, G can be represented by a set of endpoints $\mathcal{E}$ of size $2 \mid \Im \mid$, where, for each $I \in \Im$, there are unique elements $l(I)$ and $r(I)$ in $\mathcal{E}$ corresponding respectively to the left and right endpoints of I. We define an "inverse" function $\mathcal{I} : \mathcal{E} \rightarrow \Im$, which gives the corresponding interval for each member of $\mathcal{E}$. That is, if e is either $l(I)$ or $r(I)$ and $I \in \Im$ then, $\mathcal{I}(e) = I$. We shall often find it convenient to represent an interval graph as directed, so that for two overlapping intervals I_1 and I_2, $(I_1, I_2) \in E$ iff $l(I_1) < l(I_2)$.

Interval graphs are perfect graphs [36]; that is, for an interval graph $G = (V, E)$, and for every induced subgraph G' of G, the chromatic number of G' is equal to the clique number of G'.

In this chapter, we consider some parallel algorithmic issues of minimally colouring interval graphs. We use the short form "IGC" to denote the problem of minimally colouring an interval graph with a known interval representation.

If in an interval representation of an interval graph, all endpoints are not distinct, then an equivalent one of distinct endpoints can be constructed by a constant number of invocations to the All Nearest Smaller Values problem [94], provided the endpoints are sorted. We briefly outline this algorithm. Let an interval graph $G = (V, E)$, be represented by two arrays L and R containing the left and right endpoints of the intervals of G given each in a non-decreasing order, either sorted or padded-

sorted. The elements of L and R can be merged into an array A in $O(1)$ time with $n^{1+\epsilon}$ processors on a COMMON CRCW PRAM [96, 92]; if endpoints are in padded-sorted form, then the merged array will also be in padded-sorted form. Suppose there are more than one occurrences in A of an endpoint s. Say, the left most one occurs at location i and the right most one at location j. Let r (resp., t) be the nearest smaller (resp., larger) on the left (resp., right) of s in A. For, $0 \leq k \leq (j-i)$, change the endpoint of location $(i+k)$ to $s + k * \frac{t-s}{2(j-i+1)}$ if it is a right endpoint, and to $\frac{s+r}{2} + k * \frac{s-r}{2(j-i+1)}$ if it is a left endpoint. Thus, constructing the new set of endpoints, which are clearly distinct, needs only $O(1)$ time with n^2 processors, or alternatively $O(\log\log n)$ time with $n/\log\log n$ processors on a COMMON CRCW PRAM [11], and hence is in NC^1. So, in the sequel, without loss of generality we assume all $2n$ endpoints to be distinct.

6.1 IGC on Bounded Fan-in Circuits

The complexity of IGC on bounded fan-in circuits is considered in this section. It is shown that IGC may not be in NC^1 even when the left and right endpoints are separately sorted.

A linked list is a directed graph in which both the out-degree and the in-degree of a vertex can be at most one. In a monotonic linked list v can be an out-neighbour of u only if $u < v$. We will denote by "MLCC", the problem of labelling each node in a collection of disjoint monotonic linked lists, by the first element of the corresponding list; that is, MLCC is the problem of finding the connected components in a collection of disjoint monotonic linked lists.

6.1.1 A Reduction from MLCC to IGC

Consider an instance of MLCC of size n with k components, given in an array $A[1 \ldots n]$. From this instance we construct an equivalent instance of IGC as follows.

Initialise an array $B[1 \ldots 3n]$ with zeros. Replicate the lists in A in the subarray $B[n+1 \ldots 2n]$; that is, $A[i]$ is copied into $B[n+i]$, and if $A[i]$ points to $A[j]$, then a pointer is set from $B[n+i]$ to $B[n+j]$, for $1 \leq i, j \leq n$. For $n+1 \leq i \leq 2n$, if $B[i]$'s out-degree is zero, add an edge from $B[i]$ to $B[i+n]$ and if $B[i]$'s in-degree is zero, add an edge from $B[i-n]$ to $B[i]$. Now, $B[1 \ldots 3n]$ is a collection of monotonic linked lists, where each of the k non-zero elements of $B[1 \ldots n]$ (resp., of $B[2n+1 \ldots 3n]$)

is the starting point (resp., the ending point) of a list. Using prefix summation, in NC^1, compact $B[1 \ldots 3n]$ into $B[n-k+1 \ldots 2n+k]$ without disturbing the order of its elements.

Construct a set $\Im$ of intervals, so that, for $n-k+1 \leq i,j \leq 2n+k$, if $B[i]$ points to $B[j]$ then, $[i, j-\frac{1}{2}] \in \Im$. Note that both the left and right endpoints of these intervals can now be represented in sorted form in two separate arrays, say L and R.

Consider the interval graph G defined by $\Im$. Identifying each component of B with a unique colour, we get a valid colouring of G; that is, $\chi(G) \leq k$. Now, suppose that there is an optimal colouring of G that for two consecutive edges (u,v) and (v,w) of B, gives different colours to their corresponding intervals $I_u = [u, v-\frac{1}{2}]$ and $I_v = [v, w-\frac{1}{2}]$ in G. Since all endpoints are distinct, and there is no endpoint between $v-\frac{1}{2}$ and v, both I_u and I_v share $k-1$ mutually adjacent neighbours, and we get $\chi(G) = k+1 > k$, a contradiction. That is, an optimal colouring of G will use one colour per component of B. In other words, $\chi(G) = k$ and any optimal colouring of G will identify the connected components of B, and hence of A. Thus we have the following lemma.

Lemma 6.1 *An instance of MLCC of size n with k components can be reduced to an instance of IGC of size $O(n)$ and chromatic number k, in NC^1.*

6.1.2 A Reduction from IGC to MLCC

A sequential algorithm for IGC is easy to visualise. Let Q be a queue of size $\chi(G)$, which is initialised with all the available $\chi(G)$ colours. Consider the endpoints of the intervals one by one in non-decreasing order. For each left endpoint encountered, remove a colour from the front of Q and colour the corresponding interval with it, whereas, for each right endpoint, release the colour of the corresponding interval onto the back of Q. When the last of the left endpoints has been considered, the graph would have been coloured.

We attempt a direct parallelisation of this algorithm. Let $L = \{l_1, \ldots, l_n\}$ and $R = \{r_1, \ldots, r_n\}$ respectively be the sets of the left and right endpoints of the intervals given in non-decreasing order. For $1 \leq i \leq n$, let t_i be the rank of r_i in L. That is, $l_1 < \ldots < l_{t_i} < r_i < l_{t_i+1}$. So, when r_i releases the colour of the interval $\mathcal{I}(r_i)$ onto the back of Q, t_i left endpoints and i right endpoints would have been encountered and

the length of Q would be $\chi(G) - t_i + i$. Hence, the colour released by r_i will be taken up by the $(\chi(G) - t_i + i)$-th left endpoint from now on; that is, by the left endpoint $l_{t_i+\chi(G)-t_i+i} = l_{\chi(G)+i}$. In other words, both $\mathcal{I}(r_i)$ and $\mathcal{I}(l_{\chi(G)+i})$ are to get the same colour and no interval with a left endpoint between their respective left endpoints will get that colour.

Define a set of pointers $p : L \rightarrow L$ as follows: let $p(l(\mathcal{I}(r_i)))$ be $l_{\chi(G)+i}$, for $1 \leq i \leq n$. It is clear that p defines a collection of monotonic linked lists over L. Once we find the connected components in this collection we have got an optimal colouring of the graph. Thus we have the following lemma:

Lemma 6.2 *IGC can be NC^1-reduced to MLCC.*

Theorem 6.1 *IGC is equivalent to MLCC with respect to NC^1-reductions.*

6.1.3 A Reduction from List-Colouring to MLCC

A general linked list L can be visualised as being constituted of alternating stretches of forward and backward pointers in an array, and hence can be decomposed into two instances of MLCC, one consisting only of forward stretches and the other only of backward stretches. Let L' and L'' be the MLCC instances formed by the forward and backward stretches of L, respectively. An MLCC algorithm can be used to 2-colour L' as follows:

> Find the connected components in L', and identify each component with the first vertex of the component. For each vertex v in L', let L'_v be the list obtained by removing, where one exists, the edge going out of v in L'. Find the connected components in L'_v, and identify each component with the first vertex of the component. Count the vertices that are not in the same component in both L' and L'_v. Depending on whether the count is odd or even, let v have colour 1 or 2.

Similarly, 2-colour L'' also. Now, L is 4-coloured. A 4-colouring can be reduced to a 3-colouring in NC^1 [32].

From Theorem 6.1, thus, we get the following theorem:

Theorem 6.2 *The problem of 3-colouring a linked list can be reduced to IGC in NC^1.*

Linial has shown that for a linked list of length n to be 3-coloured, each of its vertices has to scan a path of length $\Omega(\log^* n)$ [72]. Besides, the problem of finding

a path between two vertices apart by a distance of $g(n)$ in a graph is not in AC^0 when $g(n) \rightarrow \infty$ with n; also, this problem is unlikely to be in NC^1 [98]. That is, an $o(\log n \log^* n)$ depth bounded fan-in circuit that 3-colours a linked list probably doesn't exist. Thus, in view of Theorem 6.2, may be, optimally colouring interval graphs is not in NC^1, after all.

6.1.4 An Upper Bound

Consider an instance L of MLCC with k components in it. Divide the input array into $n/2k$ segments of size $2k$ each. Remove the inter-segment links from L. Now, each segment forms an instance of MLCC. Use $O(\log k)$ iterations of recursive doubling on each segment to find the connected components in it. Since addresses in each segment are from a range of size $2k$ each iteration can be implemented in $O(\log k)$ time. Identify each component with the first vertex (called a root) in it. Compact the roots in each segment. Now each segment has a size of at most k. For each root v, let the nearest root, from its subsequent segments, that belongs to the same component as v be its out-neighbour; a new instance of MLCC of size at most $n/2$ is formed. Thus, in $O(\log n \log^2 k)$ time all connected components can be found. Informing each vertex of the component it belongs to, will take only an additional $O(\log n \log k)$ time.

Hence, when the number of components is restricted to $O(1)$, MLCC, is in NC^1.

From Theorem 6.2, thus, we get the following theorem, which complements the observation that 3-colouring a linked list, in general, is unlikely to be in NC^1.

Theorem 6.3 *A linked list that has at most a constant number of stretches, can be 3-coloured in NC^1.*

6.2 IGC on PRAM Models

First, we consider a problem closely related to IGC, namely, that of finding the chromatic number of an interval graph with a known interval representation.

6.2.1 Finding the Chromatic Number

Chen [16] proves that finding the chromatic number of an interval graph requires $\Omega(\log n / \log\log n)$ time, through a reduction from Parity. The proof (with

slight modifications) runs as follows. Let $A[1 \ldots n]$ be an instance of Parity. Construct $(n+1)$ intervals $I[0], \ldots, I[n]$, by setting $I[0] = [0, 2n+1]$ and for $1 \leq j \leq n$, $I[j] = [2n + j + \frac{1}{2}, 2n + j + 1]$ if $A[j] = 0$ and $I[j] = [j, 2n - j + 1]$ if $A[j] = 1$. Find the chromatic number χ of the graph corresponding to these intervals. The parity of $A[1 \ldots n]$ is even when χ is odd and odd when it is even.

However, if we use a different input representation, that is, if we assume that both the left and right endpoints are given to us in separate sorted sets, then it turns out that the chromatic number can be found in constant time.

Let an interval graph $G = (V, E)$ be represented by $L = \{l_1, \ldots, l_n\}$ and $R = \{r_1, \ldots, r_n\}$ the left and right endpoints of the intervals of G given in a non-decreasing order. The sequences L and R can be cross ranked in $O(1)$ time with $n^{1+\epsilon}$ processors [96, 92] or in $O(\log\log n)$ time with $n/\log\log n$ processors on a CREW PRAM [55]. For $1 \leq i \leq n$, let s_i be the rank of l_i in R. That is, $r_1 < \ldots < r_{s_i} < l_i < r_{s_i+1}$. Then, $i - s_i$ $(= t_i$ say), is the size of the clique formed precisely by those intervals that contain l_i. In other words, t_i is the size of the clique that $\mathcal{I}(l_i)$ forms along with its in-neighbours. Observe that any maximal clique $\mathcal{C}$, of an interval graph should contain a vertex v, such that the set of all in-neighbours of v is precisely the set $\mathcal{C} - \{v\}$. Thus, the chromatic number of G can be obtained by taking the maximum of t_i over all i. This again can be done in $O(1)$ time with $n^{1+\epsilon}$ processors [96, 92] or in $O(\log\log n)$ time with $n/\log\log n$ processors [55, 92] on a COMMON CRCW PRAM. Thus, we have:

Lemma 6.3 *The clique number of an interval graph can be found in $O(1)$ time using $n^{1+\epsilon}$ processors, $0 < \epsilon < 1$, on a COMMON CRCW PRAM, provided, the left and right endpoints of the intervals are given in sorted order, separately.*

Corollary 6.1 *The clique number of an interval graph can be found in $O(\log\log n)$ time using $n/\log\log n$ processors on a COMMON CRCW PRAM, provided, the left and right endpoints of the intervals are given in sorted order, separately.*

Even when the endpoints are only padded-sorted, for a graph with a sufficiently small chromatic number the general lower bound of $\Omega(\log n/\log\log n)$ can be surpassed. Recall that every vertex v together with its in-neighbours forms a (not necessarily maximal) clique. (See the arguments preceding Lemma 6.3.) Thus, for each

vertex $v \in V$, using $O(n)$ processors, we copy the endpoints of all of v's in-neighbours into an array $N_v[1 \ldots 2n]$, without changing their relative positions. We further add the endpoints of v to the array N_v, at their appropriate places. Now, N_v contains $2x_v \leq 2\chi(G)$ non-zero entries in non-decreasing order. Compact this array, into $2x_v$ locations in $O(\log \chi(G)/\log\log n)$ time using n processors [83], and we would have found out x_v. The chromatic number of the given graph can now be found by taking the maximum of x_v over all $v \in V$; this can be done in $O(1)$ time with $n^{1+\epsilon}$ processors, $0 < \epsilon < 1$. The overall time taken is only $O(\log \chi(G)/\log\log n)$ with $n^{1+\epsilon}$ processors.

This upper bound can be proved to be tight by giving a matching lower bound as well. We do this by showing a reduction from ordered compaction to the chromatic number problem. The ordered compaction problem [83] is defined as follows: given an array A of size n, in which only k elements are non-zero, place the k non-zero elements of A in an array B of size k, so that the elements appear in the same order in B as they do in A. Ragde [83] shows that, not only can ordered compaction problems of size n with k non-zero elements be solved in $O(\log k/\log\log n)$ time with n processors on a COMMON CRCW PRAM, but a matching lower bound, with a polynomial number of processors, also exists.

The reduction is as follows. Let $A[1 \ldots n]$ be an instance of ordered compaction that contains k non-zero elements. Create n arrays, A_i for $1 \leq i \leq n$, each of size n by setting $A_i[j] = A[j]$ when $j \leq i$ and $A_i[j] = 0$ otherwise. For, $1 \leq i \leq n$, construct an interval graph G_i on n intervals $I_i[1], \ldots, I_i[n]$, where, for $1 \leq j \leq n$, if $A_i[j] \neq 0$ then $I_i[j] = [j, 2n - j + 1]$ and if $A_i[j] = 0$ then $I_i[j] = [2n + j - \frac{1}{2}, 2n + j]$. Observe that intervals $I_i[r]$ and $I_i[s]$ overlap iff $A_i[r]$ and $A_i[s]$ are both non-zero. Find the chromatic number χ_i of G_i. Move $A[i]$ to $A[\chi_i]$. The first k locations of A now contain the non-zero elements of the input in the same order. That is, a more appropriate lower bound for finding the chromatic number of an interval graph G is $\Omega(\log \chi(G)/\log\log n)$ rather than just $\Omega(\log n/\log\log n)$.

6.2.2 A Lower Bound for IGC

From the discussion on the chromatic number problem, presented in the previous section, a lower bound of $\Omega(\log \chi(G)/\log\log n)$ on time for IGC, with a polynomial number of processors, is obvious. We prove a tighter bound: even when the left and right endpoints of the intervals are sorted separately, an $\Omega(\log n/\log\log n)$ time lower

bound holds for IGC.

Lemma 6.4 *An instance of MLCC of size n with $k = O(1)$ components can be reduced to an instance of IGC of size $O(n)$ and chromatic number k, in $O(1)$ time on CRCW PRAM using a linear number of processors.*

Proof: The proof is similar to that of Lemma 6.1, except in that instead of prefix summation, we use ordered compaction to compact $B[1 \ldots 3n]$ into $B[n-k+1 \ldots 2n+k]$. Since $k = O(1)$, the compaction takes only $O(1)$ time [83]. □

Lemma 6.5 *An instance of MLCC of size n, even when the number of components is two, requires $\Omega(\log n / \log\log n)$ time on a CRCW PRAM, with a polynomial number of processors.*

Proof: We show that Parity can be reduced to an instance of MLCC with two components. Consider an instance of Parity of size n, given in an array $X[1 \ldots n]$. Let $Y[1 \ldots 2n+2]$ be an array of vertices. Establish pointers on these vertices as follows:

$$\begin{array}{ll} \text{if } X[i] = 0 & p(Y[2i-1]) = Y[2i+1] \\ & p(Y[2i]) = Y[2i+2] \\ \text{otherwise} & p(Y[2i-1]) = Y[2i+2] \\ & p(Y[2i]) = Y[2i+1]. \end{array}$$

Now, there are exactly two monotonic linked lists in Y. Find the connected components. If $Y[2n+2]$ and $Y[2]$ are in the same component the parity is even; otherwise the parity is odd. □

Thus, we have the following theorem:

Theorem 6.4 *IGC has a lower bound of $\Omega(\log n / \log\log n)$ on time, on a CRCW PRAM, with a polynomial number of processors, even when the left and right endpoints of the intervals are sorted separately.*

6.2.3 Upper Bounds for IGC

Every step of the NC^1-reduction from IGC to MLCC, of Lemma 6.2, can be executed in $O(1)$ time on a COMMON CRCW PRAM, using a polynomial number of processors. Hence, any algorithm for MLCC indicates an algorithm for IGC with

the same asymptotic resource bounds, on a COMMON CRCW PRAM. Since MLCC is a special case of the list ranking problem, it follows that both IGC and MLCC have $O(\log n)$ time optimal algorithms [6]. An $O(\log n)$ time, n processors, EREW PRAM algorithm for IGC has been found before [89].

The list ranking problem is FL-complete for NC^1-reductions, and hence is unlikely to be in NC^1 [90]. Besides, it is known that every problem in NC^1 can be solved in $O(\log n/\log\log n)$ time with a polynomial number of processors on a CRCW PRAM [53]. Hence the following questions arise (i) whether MLCC is FL-complete for NC^1-reductions, and (ii) whether MLCC can be solved in $O(\log n/\log\log n)$ time on a CRCW PRAM with a polynomial number of processors. All these are as yet open.

However, in this section, we investigate solutions for MLCC, by imposing restrictions on the number of components (equivalently, on the chromatic number, if we consider IGC). We show that, when the number of components is limited to $O((\log n)^{1-\epsilon})$ for some $\epsilon \in (0,1)$ an instance of MLCC can be solved in $O(\log n/\log\log n)$ time with a polynomial number of processors.

The $O(\log n/\log\log n)$ time algorithm proceeds by reducing the problem size by a factor of $O((\log n)^{\epsilon})$ in every step. The following lemmas are made use of by the algorithm.

Lemma 6.6 *[22, 91] A prefix sums computation over an array of n bits can be done in $O(1)$ time using $n2^n$ processors on a COMMON CRCW PRAM, with a preprocessing time of $O(\log n/\log\log n)$.*

Proof: A single step of table-lookup is used. The input is simultaneously compared with every possible input, and the table entry corresponding to the match gives the result. Since there are 2^n possible inputs, $n2^n$ processors are required. The table can be constructed in $O(\log n/\log\log n)$ time using 2^n simultaneous copies of the standard prefix sums algorithm [22]. □

Lemma 6.7 *An instance of MLCC of size n can be solved in $O(1)$ time using $O(n2^n)$ processors on a COMMON CRCW PRAM with a pre-processing time of $O(\log n)$.*

Proof: Suppose an array $A[1\ldots n]$ contains the input. Of all the 2^n possible two-colourings of A, we choose a $\sigma : \{A[1],\ldots,A[n]\} \rightarrow \{1,2\}$ so that, for every vertex v with in-degree zero $\sigma(v) = 1$. There is a unique such two-colouring. Basically, a vertex

gets coloured 1 or 2 depending on whether its rank is odd or even counting from the beginning of its corresponding list. Hence, σ can be found in $O(1)$ time using $n2^n$ processors on a COMMON CRCW PRAM.

Make n copies $A_1[1\ldots n],\ldots,A_n[1\ldots n]$ of the input A. For the i-th copy, if $A_i[i]$ has in-degree zero, do the following. Make $A_i[0]$ the in-neighbour of $A_i[i]$ and find a two-colouring $\sigma_i : \{A_i[0],\ldots,A_i[n]\} \rightarrow \{1,2\}$ so that, every vertex with in-degree zero gets coloured 1; thus, only the vertices of the linked list that start at vertex i will get their colour changed. For $1 \leq j \leq n$, if $\sigma_i(A_i[j]) \neq \sigma(A[j])$ then let $q(A[j]) = A[i]$. Clearly $q(A[i]) = A[i]$, and we call $A[i]$ a root. Now, q defines the connected components of A; that is, vertices u and w are in same component iff $q(u) = q(w)$.

The number of processors used is $n(n+1)2^{n+1} = O(n^2 2^n)$. □

Now, the algorithm is easy to visualise and is embodied in the following theorem:

Theorem 6.5 *An instance of MLCC of size n in which the number of components is limited to $C(n)$ can be solved in $O(\frac{\log n}{\log t - \log C(n)})$ time with $O(nt2^t)$ processors, for any $t > C(n)$, on a COMMON CRCW PRAM.*

Proof: Let $A[1\ldots n]$ contain the input instance. For some $s < t$, divide A into segments of size t each. Assign $O(t^2 2^t)$ processors to each segment and solve the t-sized instance of MLCC defined by each segment in parallel in $O(1)$ time using Lemma 6.7. There can be at most $C(n)$ roots in each segment. Compact the roots of each segment in $O(1)$ time using Lemma 6.6. For each root v, let the nearest root, among its subsequent segments, that belongs to the same component as v be its out-neighbour. The compacted roots along with the new edges define an instance of MLCC of size at most $nC(n)/t$. That is in $O(1)$ time the problem size is reduced by a factor of $t/C(n)$. So, the connected components can be found in $O(\log n/\log(t/C(n))) = O(\frac{\log n}{\log t - \log C(n)})$ time. The total number of processors used is $O(t^2 2^t n/t) = O(nt2^t)$. □

Corollary 6.2 *An instance of MLCC of size n in which the number of components is limited to $C(n)$ can be solved in $O(\frac{\log n}{\log s - \log C(n)})$ time with $O(n2^s)$ processors, for any $s > C(n)$, on a COMMON CRCW PRAM.*

Proof: Set $t = s/2$ in the Theorem.

Corollary 6.3 *An instance of MLCC of size n, in which the number of components is limited to $O((\log n)^{1-\epsilon})$ for some $\epsilon \in (0,1)$ can be solved in $O(\log n / \log\log n)$ time with a polynomial number of processors on a COMMON CRCW PRAM.*

Proof: Let $C(n) = O((\log n)^{1-\epsilon})$ be the number of components in the input instance; use Theorem 2 with $t = \log n$. □

Chapter 7

Edge-Colouring of Graphs

The problem of $(\Delta+1)$-edge-colouring an arbitrary graph is not yet known to be in NC; Karloff and Shmoys [60] give an algorithm that is in NC only when Δ is at most polylogarithmic in n. This algorithm runs in

$$O(\Delta^5 \log n(\Delta + \log n + \min\{\log^3 n, \Delta^2 \log \Delta(\log^* n + \Delta^2)\}))$$

time on an EREW PRAM with $n + m$ processors. Liang, Shen and Hu [71, 70] give an $O(\Delta^{4.5} \log^3 \Delta \log n + \Delta^4 \log^4 n)$ time, $n^3 \log \Delta + n\Delta^3$ processors CRCW PRAM algorithm for $(\Delta+1)$-edge-colouring a general graph.

Fürer and Raghavachari [27] give edge-colouring algorithms that are fast, but wasteful in colours. They give an $O(\log n \log \Delta)$ time, $n+m$ processors, CREW PRAM algorithm for $c\Delta$-edge-colouring a general graph, $c > 1$, and an $O(\log^* n)$ time, $n + m$ processors, CREW PRAM algorithm for Δ^2-edge-colouring a general graph.

In this chapter, we improve these results: the following algorithms for edge-colouring a general graph are presented:

- an algorithm that finds a $(\Delta + d)$-edge-colouring, $1 \leq d < \Delta$, in $O((\log d + (\Delta/d)^4) \log^2 n)$ time, using $n + m$ processors
- an algorithm that finds a $\Delta^{1+\epsilon}$-edge-colouring, $0 < \epsilon < 1$, in $O(\log \Delta \log(\log^* n))$ time, using $n\Delta^{1+\epsilon}$ processors

Also, we show that from Theorem 5.2 it would follow that a cubic graph can be 4-edge-coloured in $O(\log n)$ time with $O(n)$ operations on an EREW PRAM.

7.1 $(\Delta + d)$-Edge-Colouring, $d \geq 1$

An $O((\log d + (\Delta/d)^4) \log^2 n)$ time, $n+m$ processors EREW PRAM algorithm for $(\Delta + d)$-edge-colouring an arbitrary graph, $1 \leq d < \Delta$, is presented in this section. (The basic idea of this algorithm was discussed, in the context of $(\Delta+1)$-edge-colouring of a bounded degree graph, in the author's M. Tech. thesis [85]; see also [86].)

The algorithm is based on the proof of Vizing's theorem by Misra and Gries [76]. Besides being faster than the

$$O(\Delta^5 \log n(\Delta + \log n + \min\{\log^3 n, \Delta^2 \log \Delta(\log^* n + \Delta^2)\}))$$

time algorithm of [60], our algorithm has a simpler analysis.

7.1.1 A Proof of Vizing's Theorem

Here we briefly sketch a proof of Vizing's theorem. It is essentially similar to the one by Misra and Gries [76], but some of the details are altered to suit for our purposes better. (For another proof see [26, 13].) The proof uses induction on the number of edges. Basis: every graph with Δ edges is $(\Delta + 1)$-edge-colourable. Hypothesis: every graph with $m - 1$ edges is $(\Delta + 1)$-edge-colourable.

Let $G = (V, E)$ be a graph with m edges. For $e = \{v_0, v_1\} \in E$, obtain a $(\Delta + 1)$-edge-colouring of $G - e$; this is possible, by the hypothesis. A colour c is said to be free at $v \in V$ if none of the edges incident at v is coloured c. Clearly, at least one colour is free at each vertex of G. Assume that v_0 and v_1 do not have a common free colour; otherwise, e can be given that colour.

Let a (c_0, c_k)-fan $F(v_0) = (v_0, c_0, v_1, c_1, \ldots, v_k, c_k)$, for colours $c_0, ..., c_k$ and distinct $v_1, ..., v_k \in N(v_0)$, be a data structure that satisfies the following conditions:

- the colour c_i is missing at v_i for $0 \leq i \leq k$.
- the edge (v_0, v_i) is coloured c_{i-1} for $1 < i \leq k$.
- either c_k is missing at v_0 or $c_k = c_j$ for some j such that $0 < j < k$.

Vertices $v_0 = h(F)$ and $v_k = t(F)$ are respectively called the *head* and *tail* of the fan $F = F(v_0)$.

Since, for any colour, there is at most one edge of that colour incident with any vertex, a colour other than c_k can occur only once in the fan, whereas c_k can occur

at most twice. If c_k is missing at v_0 (that is c_k occurs only once in the fan or $c_k = c_0$; in this case the fan is called *local*), then for $1 \leq i \leq k$, the edge $\{v_0, v_i\}$ can be given the colour c_i, thus completing the proof. (This is called the *fan operation*). So, assume that the fan is not local; c_k occurs twice in the fan, and $c_k \neq c_0$; $c_k = c_j$ for some j such that $0 < j < k$. Let $v_j = m(F)$ be called the *middle* of F.

In an edge coloured graph any bichromatic component must be a path or a cycle. (A c_0-c_k component is a maximal connected subgraph induced by edges coloured c_0 or c_k.)

The head and tail of F are the only vertices in F at which colours c_0 or c_k are free. and hence are the only vertices at which a c_0-c_k path can end. Since, in addition, F is a non-local fan, a c_0-c_k path indeed ends at each of them. These three c_0-c_k paths need not all be distinct. But, at least two of them are distinct, because, at least two paths are needed to account for three endpoints.

Let $H(F)$, $M(F)$ and $T(F)$ denote the c_0-c_k paths ending at $h(F)$, $m(F)$ and $t(F)$ respectively. Define $e(F)$, the *elect* of F, as follows: if $H(F) = M(F)$, then let $e(F) = t(F)$, otherwise, let $e(F) = m(F)$. Also, let $E(F) = M(F)$ or $E(F) = T(F)$ depending on whether $e(F) = m(F)$ or $e(F) = t(F)$.

Interchange colours c_0 and c_k in either $H(F)$ or $E(F)$, but not both. (This is the *chain operation*).

If $e(F) = t(F)$ and the chain operation was performed on $E(F)$, then $(v_0, c_0, v_1, c_1 \ldots, v_k, c_0)$ is now a valid fan, and it is local.

If $e(F) = t(F)$ and the chain operation was performed on $H(F)$ then $(v_0, c_k, v_1, c_1 \ldots, v_k, c_k)$ is now a valid fan, and it is local.

If $e(F) = m(F)$ and the chain operation was performed on $E(F)$ then $(v_0, c_0, v_1, c_1 \ldots, v_j, c_0)$ is now a valid fan, and it is local.

If $e(F) = m(F)$ and the chain operation was performed on $H(F)$ then $(v_0, c_k, v_1, c_1 \ldots, v_j, c_k)$ is now a valid fan, and it is local.

A fan operation will now complete the proof.

Observation 7.1 *Of the two distinct colours free at v_1, only one has been used in the proof. Hence the proof would be valid even if v_1 had only one free colour.*

7.1.2 Creating Fans in Parallel

Consider a graph $G = (V, E)$ that is partially $(\Delta+1)$-edge-coloured so that any two uncoloured edges of G are at least three edges apart. In this graph, if fan operations are performed at all uncoloured edges simultaneously, no two of them will interfere with each other.

We now show how to construct fans at the uncoloured edges of G. We assume that G is given in adjacency list representation; thus, each vertex has a list $N(v_0)$ of all its neighbours.

Let $e = \{v_0, v_1\}$, be an uncoloured edge; arbitrarily choose one endpoint, say v_0, of e as the head of the fan $F(v_0)$ to be constructed.

Step 1: Form a directed graph $H'(v_0)$ on $\{v_0\} \cup N(v_0)$ as follows: Let (v_0, v_1) be an edge in $H'(v_0)$. For all $u \in N(v_0)$, let $\sigma(u)$ be a colour that is missing at u; if there is more than one colour missing at u, then we choose an arbitrary one of them as $\sigma(u)$. For $w \in N(v_0)$, if the edge $\{v_0, w\}$ in G is coloured $\sigma(u)$ then let (u, w) be an edge in $H'(v_0)$. Clearly, the out-degree of every vertex in $H'(v_0)$ is at most one.

REMARK: Assume that each chosen head v of a fan has its adjacency list sorted on colours, and hence has its incident edges in an array $Z(v)$ indexed by colours. With processors assigned one per neighbour of v_0, each $u \in N(v_0)$, can find out its out-neighbour in $H'(v_0)$ in $O(1)$ time by performing a concurrent read from $Z(v_0)[\sigma(u)]$. On an EREW PRAM, this concurrent read can be simulated in $O(\log \Delta)$ time [55]. The edge-lists, along with twin pointers, of $H'(v_0)$, and hence of the underlying undirected graph $H(v_0)$, can be formed in an additional $O(\log \Delta)$ time on an EREW PRAM.

Step 2: Each component of $H'(v_0)$ is a pseudo tree; there is at most one directed cycle in each component. Let C denote the component that contains v_0 in $H'(v_0)$, and let C' denote the cycle in C. To find a fan at v_0, we need only to find the set of vertices reachable from v_0 in C. Find C and C' using the Euler Traversal Technique of [95]; $C - C'$ is a rooted forest. Use the parallel tree contraction technique [1] to find the set of vertices reachable from v_0 in $C - C'$; this set together with C' form the set of all vertices reachable from v_0 in C.

REMARK: Note that $H'(v_0)$ and the underlying undirected graph $H(v_0)$ have been formed in adjacency list representation, and that twin pointers are available; that is,

the adjacency list entries $[x, y]$ and $[y, x]$ corresponding to an edge $\{x, y\}$ in $H(v_0)$ point to each other. Suppose next($[x, y]$) is the entry following $[x, y]$ in the adjacency list of x. Then the traversal T defined by the pointers "tour-next" is constructed as follows:

for each adjacency list entry of $H(v_0)$ pardo

tour-next($[x, y]$) = next($[y, x]$)

Each component of $H(v_0)$ that has a cycle, contributes two lists to the traversal T. One (say L') of the two lists contributed by C corresponds to C'—this list does not traverse v_0. The other list (say L'') contributed by C corresponds to $C - C'$ and hence traverses v_0. Using list ranking, identify L'' among the lists of T; this is easy, because L'' contains v_0. Since L' is the only list in T that has vertices (of $H(v_0)$) in common with L'', L' can be now identified in $O(1)$ time on a CRCW PRAM, and hence in $O(\log \Delta)$ time on an EREW PRAM with Δ processors [55].

Now that $C - C'$ has been identified, label v_0 with 1 and every other leaf of $C - C'$ with 0. For each node of $C - C'$, find the maximum among all its descendant leaves; all nodes for which the result is 1, along with C', belong to the subgraph C'' of C that is induced by vertices reachable from v_0 through a directed path. By [1] this takes $O(\log \Delta)$ time on an EREW PRAM.

Step 3: Clearly, the subgraph C'' induced by the set of vertices reachable from v_0 in C is either a simple directed path, or a simple directed path augmented by a vertex disjoint directed cycle. Also, C'' gives a fan with v_0 as its head. When C'' is a simple directed path, the fan is local too; the last vertex of the path is the tail of the fan. If the fan is not local, then the vertex with in-degree two in C'' (there is exactly one such vertex) is pointed to by the middle and tail of the fan.

Thus, we have the following lemma:

Lemma 7.1 *For a partially $(\Delta+1)$-edge-coloured graph, where between any two uncoloured edges of G there is no path of length less than three, a fan can be constructed at each uncoloured edge in parallel in $O(\log \Delta)$ time with $O(n + m)$ processors on an EREW PRAM.*

7.1.3 The Algorithm

Consider a graph $G = (V, E)$ that is partially edge-coloured using the palette $\{1, \ldots, \Delta + 1\}$, so that between any two uncoloured edges of G there is no path of

length less than three. Let E' be the set of uncoloured edges in G.

Let the *class* of any fan F to be constructed in G be the ordered pair (c,d), $c < d$, if F is either a (c,d)-fan or (d,c)-fan. So, there would be $O(\Delta^2)$ classes of fans in G at any time. The set of fans belonging to class (c,d), will be denoted by $\mathcal{C}_{cd}$.

The following procedure assigns a colour from $\{1,\ldots,\Delta+1\}$ to to each edge of E':

Procedure Solve(E')

Loop through the following steps:

Step 1: Construct fans at every uncoloured edge in G. Perform a fan operation on every local fan constructed.

Step 2: Of the fan-classes of G, find the class with the maximum cardinality; let that class be (c,d).

Step 3: Form a conflict graph $\mathcal{H}$ as follows: corresponding to each c-d path of G, let there be a vertex in $\mathcal{H}$, and let two vertices x and y of $\mathcal{H}$ be adjacent iff the c-d paths corresponding to x and y end at the same fan in G. A c-d path P in G is said to end in a fan F, if P is either $H(F)$ or $E(F)$.
REMARK: Each vertex of $\mathcal{H}$ can have (at most) two neighbours; a c-d path may end in a fan at both its endpoints. So, $\mathcal{H}$ consists of chains and cycles. Note that each fan in $\mathcal{C}_{cd}$ corresponds to an edge in $\mathcal{H}$.

Step 4: From every odd cycle of $\mathcal{H}$ remove one vertex. Rank the remaining graph, and for vertices of odd rank swap colours c and d in the corresponding c-d paths of G.
REMARK: Clearly, colours are swapped in at least two thirds of the c-d paths of G. Hence, at least two thirds of what were (c,d) or (d,c)-fans have now become (c,c) or (d,d)-fans, which are clearly local. (See the proof of Vizing's theorem in Section 7.1.1.) For a fan $F \in \mathcal{C}_{cd}$, if there are only two distinct c-d paths among $H(F)$, $M(F)$ and $T(F)$, then colours c and d have been swapped in at most one of those two paths. On the other hand, if $H(F)$, $M(F)$ and $T(F)$ are all distinct, and hence, $E(F) \neq T(F)$, then $T(F)$ also may have had its colours swapped, along with at most one of $H(F)$ and $M(F)$; but, this does affect the new fan obtained, because, it does not contain $t(F)$.

Step 5: Perform a fan operation on every local fan in G.

REMARK: At least $2/3 * |\mathcal{C}_{cd}|$ fans are now solved. But, swapping colours c and d in some c-d paths of G may have destroyed some fans not in $\mathcal{C}_{cd}$.

Lemma 7.2 *The procedure call "SOLVE(E')" assigns a colour from $\{1,\ldots,\Delta+1\}$ to each edge of E' in $O(\Delta^2 \log^2 n)$ time with $n+m$ processors on an EREW PRAM.*

Proof: By Lemma 7.1, Step 1 of the loop takes $O(\log \Delta)$ time. The rest of the steps are dominated by an instance each of sorting $O(n)$ items and ranking a list of $O(n)$ length. Hence, clearly, each iteration of the loop can be executed in $O(\log n)$ time time with $n+m$ processors on an EREW PRAM. Each iteration reduces the total number of uncoloured edges by a fraction of $O(1/\Delta^2)$. In other words, each iteration reduces the number of uncoloured edges to $O((\Delta^2-1)/\Delta^2)$ of what had been there before. So, the number of iterations required to colour all edges is $O(\log n/(\log(\Delta^2/(\Delta^2-1)) = O(\Delta^2 \log n)$. Hence the lemma. □

Now we present a procedure that edge-colours a general Graph:

Procedure Edge-Colour($G, r, \Delta(G)$)

Input: A graph G; r is a parameter, $r < \log \Delta(G)$.
Output: A $\Psi(r,\Delta(G))$-edge-colouring of G; $\Psi(r,\Delta) \geq \Delta+1$.

Step 0: If $\Delta(G) = 2$, then G is collection of lists and cycles. Obtain a 3-edge-colouring of G. **exit.**

Step 1: Decompose the edge set E of G into two disjoint sets E_0 and E_1 so that
(a) in $G_0 = (V, E_0)$, for some $s \in V$, every vertex other than s has a degree of at most $\lceil \frac{\Delta(G)}{2} \rceil$, and s has a degree of at most $\lceil \frac{\Delta(G)}{2} \rceil + 1$, and
(b) in $G_1 = (V, E_1)$ every vertex has a degree of at most $\lceil \frac{\Delta(G)}{2} \rceil$.
REMARK: The details of this step are given later in this chapter.

Step 2: For an edge $e = \{s,t\}$ in G_0, let $G_0' = G_0 - e$. Now both G_0' and G_1 are $\lceil \frac{\Delta(G)}{2} \rceil$-degree graphs.
Execute the following two procedure calls in parallel:

Call Edge-Colour($G_0', r-1, \lceil \frac{\Delta(G)}{2} \rceil$).

Call Edge-Colour($G_1, r-1, \lceil\frac{\Delta(G)}{2}\rceil$).

REMARK: Now both G'_0 and G_1 are $\Psi(r-1, \lceil\frac{\Delta(G)}{2}\rceil)$-edge-coloured.

Step 3: Extend the edge-colouring of G'_0 to G_0 (e is the only uncoloured edge of G_0) as follows: create a fan for e with t as its head; perform fan and chain operations, as necessary, and give e a colour in $\{1,\ldots,\Psi(r-1,\lceil\frac{\Delta(G)}{2}\rceil)\}$. Note that by Observation 7.1 this is feasible. Now both G_0 and G_1 are $\Psi(r-1,\lceil\frac{\Delta(G)}{2}\rceil)$-edge-coloured.

Add $\Psi(r-1,\lceil\frac{\Delta(G)}{2}\rceil)$ to every colour used in G_1.

REMARK: Now G is $2\Psi(r-1,\lceil\frac{\Delta(G)}{2}\rceil)$-edge-coloured. For $1 \leq x \leq 2\Psi(r-1,\lceil\frac{\Delta(G)}{2}\rceil)$, let E_x be the set of edges now coloured x in G.

Step 4: If $r \geq 1$, then skip this step; otherwise, proceed. For $\Delta(G)+1 < x \leq 2\Psi(r-1,\lceil\frac{\Delta(G)}{2}\rceil)$, uncolour the edges of E_x.

For each colour x in $\{\Delta+2,\ldots,2\Psi(r-1,\lceil\frac{\Delta}{2}\rceil)\}$ do:

> Form a conflict graph $C = (E_x, \mathcal{E})$ where $\{e,f\} \in \mathcal{E}$ for some $e,f \in E_x$ iff there is a path of length less than 3 between e and f in G. The maximum vertex degree α of C is $O(\Delta^2)$; note that there can be at most one edge of colour x at any vertex. Find an $(\alpha+1)$ vertex-colouring σ of C. Let E_x^j denote the set of edges e in E_x such that $\sigma(e) = i$. For $1 \leq j \leq \alpha+1$, in turn call SOLVE(E_x^j).

Now we describe Step 1 in detail:

Substep 1.1: If G is not Eulerian, then add a vertex u that is adjacent to every odd degree vertex in G. Let G' be the augmented graph thus obtained. If G is Eulerian let $G' = G$.

REMARK: Since the number of odd degree vertices in any graph is even, G' is Eulerian.

Substep 1.2: Find an Euler tour in G'. If $G \neq G'$ let this tour start at $z = u$, otherwise, let it start at some $z \in V$. Label the edges of the tour 0 and 1 alternately; the first edge is labelled 0. Let G_0 and G_1 be the subgraphs of G' induced by the edges labelled 0 and 1 respectively.

REMARK: If G' has an even number of edges, then clearly, G_0 and G_1 are $\lceil\frac{\Delta(G)}{2}\rceil$-degree graphs. But if G' has an odd number of edges, then both the first and the last edges of the Euler tour obtained would be labelled 0, and hence, z would have incident to it

two more label-0 edges than label-1 edges. If $G \neq G'$ and hence $z \notin V$, then this would not matter, because neither G_0 nor G_1 would contain z, and G_0 and G_1 would still be $\lceil \frac{\Delta(G)}{2} \rceil$-degree graphs.

If $G = G'$ and G' has an odd number of edges, then the vertex z has degree $\frac{\Delta(G)}{2} + 1$ in G_0 and $\frac{\Delta(G)}{2} - 1$ in G_1. Every vertex other than z has degree $\frac{\Delta(G)}{2}$ in both G_0 and G_1.

Step 4 is executed only when $r < 1$. Clearly, the number of colours used by the procedure call "Edge-Colour$(G, 0, \Delta(G))$" is $\Psi(0, \Delta(G)) = \Delta(G) + 1$.

So, the number of colours used by the procedure call "Edge-Colour$(G, r, \Delta(G))$" is

$$\begin{aligned}\Psi(r,\Delta) &= 2\Psi\left(r-1, \left\lceil \frac{\Delta(G)}{2} \right\rceil\right) = 4\Psi\left(r-2, \left\lceil \frac{\Delta(G)}{4} \right\rceil\right) = \cdots \\ &= 2^r \Psi\left(0, \left\lceil \frac{\Delta(G)}{2^r} \right\rceil\right) = 2^r \left(\left\lceil \frac{\Delta(G)}{2^r} \right\rceil + 1\right)\end{aligned}$$

Thus, for $d \geq 1$, the procedure call "Edge-Colour$(G, \lfloor \log d \rfloor - 1, \Delta)$" gives a $(\Delta + d)$-edge-colouring of G.

Let $T(r, \Delta)$ be the time taken by the procedure call "Edge-Colour$(G, r, \Delta(G))$". A 3-edge-colouring of a 2-degree graph can be found (Step 0) in $O(\log(\log^* n))$ time with n processors on an EREW PRAM [48]. An Euler tour in an Eulerian graph can be found (Step 1) in $O(\log n)$ time on a CRCW PRAM [8], and hence, in $O(\log^2 n)$ time on an EREW PRAM, with $O(n+m)$ processors. Step 2 takes $T(r, \lceil \frac{\Delta}{2} \rceil)$ time. Step 3 takes $O(\log n)$ time with $n + m$ processors on an EREW PRAM. An $(\alpha + 1)$-colouring of an α-degree graph can be found in $O(\alpha \log \alpha (\log^* n + \log \alpha)$ time, with n processors, on an EREW PRAM [32]. Hence, by Lemma 7.2, Step 4 takes $O(\Delta^2 \log \Delta (\log^* n + \log \Delta) + \alpha \Delta^2 \log^2 n) = O(\Delta^4 \log^2 n)$ time with n processors on an EREW PRAM. Step 4 is executed only when $r < 1$. Thus, after r becomes zero, Step 4 dominates the time taken, and hence, $T(0, \Delta) = O(\Delta^4 \log^2 n)$. Therefore, for $r \geq 1$, for an appropriate constant c,

$$\begin{aligned}T(r,\Delta) &= T\left(r-1, \left\lceil \frac{\Delta}{2} \right\rceil\right) + c\log^2 n \\ &< T\left(r-2, \left\lceil \frac{\Delta}{4} \right\rceil\right) + 2c\log^2 n \\ &< \cdots \\ &< T\left(0, \left\lceil \frac{\Delta}{2^r} \right\rceil\right) + rc\log^2 n\end{aligned}$$

$$= O((\Delta/2^r)^4 \log^2 n + r \log^2 n)$$

Setting $r = \lfloor \log d \rfloor - 1$, thus we have the following theorem:

Theorem 7.1 *A graph can be $(\Delta + d)$-edge-coloured in $O((\log d + (\Delta/d)^4) \log^2 n)$ time with $O(n + m)$ processors on an EREW PRAM.*

7.2 $\Delta^{1+\epsilon}$-Edge-Colouring, $0 < \epsilon < 1$

An $O(\log \Delta \log(\log^* n))$ time, $n\Delta^{1+\epsilon}$ processor EREW PRAM algorithm for $\Delta^{1+\epsilon}$-edge-colouring an arbitrary graph is now presented. We employ a technique similar to the one that Gabow and Kariv [28], and Lev, Pippenger and Valiant [69] use to $2^{\lceil \log \Delta \rceil}$-edge-colour a bipartite graph.

Consider a graph $G = (V, E)$ with $V = \{1, \ldots, n\}$. Assume that G is given in adjacency list representation. Let $N(v)$ be the adjacency list of $v \in V$, and $[v, w]$ the entry in $N(v)$ corresponding to an edge $\{v, w\} \in E$.

Visualise each edge of G as having two halves, one half for each end vertex. Suppose each vertex of G has a palette $\{0, \ldots, \Delta - 1\}$ of colours. Let each vertex assign, from its palette, a unique colour to every half-edge incident with it. This can be done as follows. For each $v \in V$, in parallel, rank $N(v)$, starting from zero. Let $r_v(w)$ be the rank of $[v, w]$ in $N(v)$; $r_v(w) \leq \Delta - 1$. For an edge $\{v, w\} \in E$, let $\sigma_0(v, w) = r_v(w)$ be the colour assigned to $\{v, w\}$ by v.

Thus, each edge receives two colours, one for each half; these two colours need not be the same. We now proceed to modify the present colouring so that both halves of an edge become coloured the same; but this will cost us some extra colours.

For an $n \times \Delta$ array $A_0[1 \ldots n, 0 \ldots \Delta - 1]$, let $A_0[v, \sigma_0(v, w)] = [v, w]$. Let b and B, $b < B < \Delta$, be two parameters to be chosen later. Suppose the colours in the palette are represented as base-b integers. Such a representation will have $\lceil \frac{\log \Delta}{\log b} \rceil$ digits.

The algorithm now proceeds in $\lceil \frac{\log \Delta}{\log b} \rceil$ phases. In each phase, at a high level, for each edge e of G, the least significant digits of the two colours on edge e, which are two possibly different base-b numbers, are replaced by a base-B number, the same in both cases. Also, at the end of each phase, we perform a "shift-right-with-rotate" on every colour of G, so that each phase deals with a different digit of the original

colouring. After $\lceil \frac{\log \Delta}{\log b} \rceil$ phase, the graph would be validly edge-coloured, but each of the $\lceil \frac{\log \Delta}{\log b} \rceil$ digits of each colour will be a base-B number.

Now we describe the phases more formally. With respect to the phases, we maintain the following invariants: at the beginning of the i-th phase, $1 \leq i \leq \lceil \frac{\log \Delta}{\log b} \rceil$, G is $(B/b)^{i-1}\Delta$-edge-coloured; the colours are from $\{0, \ldots, (B/b)^{i-1}\Delta - 1\}$. For a colour c, let $R_{i-1}(c)$ be a $\lceil \frac{\log \Delta}{\log b} \rceil$-digit representation of c, where the $i-1$ most significant digits are all in base B and all lesser significant digits are in base b. For an edge $\{v, w\} \in E$, let $\sigma_{i-1}(v, w)$ denote the colour assigned to $\{v, w\}$ by v, at the beginning of the i-th phase. Colours $\sigma_{i-1}(v, w)$ and $\sigma_{i-1}(w, v)$ agree on the $(i-1)$ most significant digits. The adjacency list of G is given in an $n \times (B/b)^{i-1}\Delta$ array $A_{i-1}[1 \ldots n, 0 \ldots (B/b)^{i-1}\Delta - 1]$; $A_{i-1}[v, x]$ contain $[v, w]$ iff $x = \sigma_{i-1}(v, w)$. Here, we do not assume any specific order of linking for entries in an adjacency list. Note that the invariants are satisfied at the beginning of the first phase.

The i-th phase, $1 \leq i \leq \lceil \frac{\log \Delta}{\log b} \rceil$, is now described:

Phase i

Step 1: Divide the set of $(B/b)^{i-1}\Delta$ presently used colours into classes of size b each; a colour c is in the j-th class if $\lfloor c/b \rfloor = j$. Two colours c_1 and c_2 are in the same class iff $R_{i-1}(c_1)$ and $R_{i-1}(c_2)$ agree on all but the least significant digit. There are $Q_i = (B^{i-1}/b^i)\Delta$ classes.

Form a set V_i of vertices as follows: for each $v \in V$, and $0 \leq j \leq Q_{i-1}$, let $v_{i,j}$ be in V_i; that is, V_i has Q_i copies of each member of V.

Construct a graph G_i on V_i as follows: for each adjacency list entry $[v, w]$ of G, if $\sigma_{i-1}(v, w)$ belongs to the j-th class and $\sigma_{i-1}(w, v)$ belongs to the k-th class, then place an edge between $v_{i,j}$ and $w_{i,k}$.

REMARK: There is a one to one correspondence between the edges of G and G_i. Also, two vertices $v_{i,j}$ and $w_{i,k}$ are adjacent in G_i only if $\sigma_{i-1}(v, w)$ belongs to the j-th class. Thus, G_i is a b-degree graph. Since, all the edges of class j (an edge is said to be in the same class as its colour) incident at $v \in V$, appear in a subarray of size b of the v-th row of A_{i-1}, the adjacency lists of G_i can be constructed in $O(\log b)$ time on an EREW PRAM, provided a processor is stationed at each cell of A_{i-1}. Note that for

each adjacency list entry $[v,w]$ of G, the class number of $\sigma_{i-1}(w,v)$ can be determined in $O(1)$ time.

Step 2: Edge-colour G_i using colours from $\{0,\ldots,B-1\}$. We assume that there is a $T(B,b)$ time algorithm for B-edge-colouring a b-degree graph. Suppose, the edge $\{v_{i,j},w_{i,k}\}$ gets colour l. Let $\sigma_i(v,w)=\langle l,j\rangle$ and $\sigma_i(w,v)=\langle l,k\rangle$.
REMARK: Clearly, $R_{i-1}(\sigma_i(v,w))$ and $R_{i-1}(\sigma_i(w,v))$ agree on the i-most significant digits. The total number of colours present in G now is $(B/b)^i\Delta$.

Step 3: Copy the adjacency lists of G into $A_i[1\ldots n,0\ldots(B/b)^i\Delta-1]$, an $n\times\Delta(B/b)^i$ array: if $\sigma_i(v,w)=x$, then let $A_i[v,x]$ contain $[v,w]$.

The algorithm described above finds a $\Psi(B,b)=B^{\lceil\log\Delta/\log b\rceil}$-edge-colouring in

$$O\left(\frac{\log\Delta}{\log b}(T(B,b)+\log b)\right)$$

time with $n\Psi(B,b)$ processors on an EREW PRAM; here $T(B,b)$ is the time required to B-edge-colour a b-degree graph.

Let $G_L=(E,\mathcal{E})$ be the line graph of G; each edge of G corresponds to a vertex of G_L, and two vertices of G_L are adjacent iff the corresponding edges of G are adjacent in G. Any k-vertex-colouring of G_L is a k-edge-colouring of G. Since each edge of G is adjacent to at most $\Delta(G)-1$ edges at either end, G_L has a maximum vertex degree of $2\Delta(G)-2$. Thus, a $(2\Delta(G)-1)$-edge-colouring of G can be found in $O(\Delta\log\Delta(\log(\log^* n)+\Delta))$ time with $n+m$ processors [32] on an EREW PRAM.

Hence, a $\Psi(2b,b)=(2b)^{\lceil\log\Delta/\log b\rceil}$-edge-colouring of an arbitrary graph can be found in $O(b\log\Delta(\log(\log^* n)+b)$ time with with $n\Psi(2b,b)$ processors on an EREW PRAM. Note that $\Psi(2b,b)=(2b)^{\lceil\log\Delta/\log b\rceil}$ is at most $\Delta^{1+2/\log b}$, for $b<2^{\sqrt{\log(\Delta-1)}-1}$.

Let $b=2^{2/\epsilon}$; $0<\epsilon<1$; then, $\epsilon=2/\log b$. Thus, we have the following theorem:

Theorem 7.2 *A graph can be $\Delta^{1+\epsilon}$-edge-coloured in $O(\log\Delta\log(\log^* n))$ time with $n\Delta^{1+\epsilon}$ processors on an EREW PRAM; $0<\epsilon<1$.*

7.3 4-Edge-Colouring a 3-degree Graph

Let $G = (V, E)$ be a 3-degree graph. Each edge of G is adjacent to at most 4 edges, at most two at either of its end-vertices. Thus, the line graph $G_L = (E, \mathcal{E})$ has a maximum vertex degree of 4. Moreover, G_L cannot have a 5-clique; this is because, five mutually adjacent vertices in G_L must correspond to five mutually adjacent edges in G, which must then all be incident with the same vertex of G. Hence, G_L is a 4-degree Brooks' graph. Obtain a 4-vertex colouring of G_L, and this would correspond to a 4-edge colouring of G.

By Corollary 5.2, we get the following theorem.

Theorem 7.3 *A 3-degree graph can be 4-edge-coloured in $O(\log n)$ time with $\frac{n}{\log n}$ processors on an EREW PRAM.*

The above theorem improves the result of [60] where a 3-degree graph is 4-edge-coloured in $O(\log n)$ time with n processors on a CRCW PRAM.

Chapter 8

Conclusions

Several parallel algorithmic issues related mainly to sorting, merging and graph colouring were considered in this thesis.

In chapter 2, we addressed the question, whether k-way merging is exactly as hard as sorting k numbers and merging 2 arrays of n elements with the same processor advantage in each case, i.e.,

$$k\_\text{WAY\_MERGE}(n,p) \stackrel{?}{=} \Theta(\text{SORT}(k, \tfrac{pk}{n}) + \text{MERGE}(n,p)) \tag{8.1}$$

We presented an algorithm schema for k-way merging. For comparison-based versions of sorting, merging and k-way merging, using the schema we showed that Eq. 8.1 is satisfied, on all of EREW, CREW and CRCW PRAM models. For integer versions, on a CRCW PRAM, when we use the best known integer sorting and merging algorithms in our schema, and the keys are integers of $(\log n)^{\Omega(1)}$ bits, again, Eq. 8.1 is satisfied. However, for improvements to be anticipated in sorting and merging algorithms, as well as, for smaller integer keys, our schema may not satisfy Eq. 8.1. But, we conjecture that the equation will nevertheless hold. Theorem 2.12 can be seen as an evidence in favour of this conjecture.

A merge sorting algorithm that runs in $\frac{\log n}{\log \alpha} \cdot 2^{O(\log^*(\log n/\log \alpha))}$ steps, with $n\alpha$ processors, $2 \leq \alpha \leq n$, on the parallel comparison model was presented in Chapter 3. This algorithm comes close to achieving the $\Omega(\frac{\log n}{\log \alpha})$ lower bound on sorting on the parallel comparison model; moreover it is very different from Cole's merge sort. It remains to be seen if this algorithm can be improved into matching the lower bound. On a CREW PRAM, a corresponding question would be whether $O(\log n)$ time could

be achieved, thereby giving a simple sorting algorithm that does not use pipelining. A still harder problem would be designing a simple $O(\log n)$ time optimal comparator network for sorting.

An $O((\log\log n)\log^*(\log^* n))$ time optimal algorithm on the TOLERANT CRCW PRAM, for 3-colouring general rooted forests, was presented in Chapter 4. Here optimality was achieved at a loss of speed: a rooted forest can be 3-coloured in $O(\log(\log^* n))$ time on an n processor CREW PRAM. Moreover, our algorithm depends on the approximate compaction problem that has a parallel complexity of $\Theta(\log\log n)$. It remains an open question whether an $o(\log\log n)$ time optimal algorithm for 3-colouring general rooted forests can be found.

In Chapter 4, we also show that a TOLERANT PRAM of size N with a linear address space, can be slowed down by any factor $\lambda = \Omega(\log\log N)$, with no asymptotic increase in space or cost. Unlike other CRCW PRAM models, TOLERANT is not known to be self-simulating in general [39]; and probably it is not—but this remains to be proved.

An $O(\log n)$ time optimal EREW PRAM algorithm for Brooks' colouring of bounded degree graphs was presented in Chapter 5. In addition, a combinatorial result was obtained as evidence to suggest that an asymptotically faster algorithm may exist. But, how to use this combinatorial result in an algorithm is not clear yet. The lower bound problem on Brooks' colouring is also open; the trivial $\Omega(\log(\log^* n))$ time lower bound on the distributed model is the only one currently known.

In Chapter 6, we argued that the problem of colouring an interval graph with a known interval representation (IGC) may not be in NC^1. In this context, the following questions arise (*i*) whether IGC is FL-complete for NC^1-reductions, (*ii*) whether IGC can be solved in $O(\log n/\log\log n)$ time on a CRCW PRAM with a polynomial number of processors. Both of these are as yet open. However, we gave an $O(\log n/\log\log n)$ time, polynomial processors, algorithm for IGC, for the special case of the chromatic number being $O((\log n)^{1-\epsilon})$, $0 < \epsilon < 1$.

Two parallel algorithms for wasteful edge colouring were presented in Chapter 7. Finding a minimal edge colouring is computationally difficult even sequentially; The problem of $(\Delta+1)$-edge-colouring a general graph, on the other hand, has a polynomial time sequential solution, but, does not render itself to easy parallelisation; it has been an open question for long whether this problem is in NC or not.

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c.\log n$ parallel steps. *Combinatorica*, 3:1–48, 1983.

[4] S. G. Akl, M. Cosnard, and A. G. Ferreira. Data-movement-intensive problems: two folk theorems in parallel computation revisited. *Theoretical Computer Science*, 95:323–337, 1992.

[5] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7:576–583, 1986.

[6] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6(6):859–868, 1991.

[7] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proceedings, 27th ACM Symposium on Theory of Computing*, pages 427–436, 1995.

[8] M. Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29:330–337, 1984.

[9] Y. Azar and U. Vishkin. Tight comparison bounds on the complexity of parallel sorting. *SIAM Journal of Computing*, 16:458–464, 1987.

[10] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of ACM*, 36(3):643–670, 1989.

[11] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14:344–370, 1993.

[12] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. *Information and Computation*, 94:29–47, 1991.

[13] B. Bollobas. *Graph Theory, An Introductory course*. Springer-Verlag, 1979.

[14] A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parllel models of computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[15] S. Chaudhuri. Sensitive functions and approximate problems. *Information and Computation*, 126:161–168, 1996.

[16] L. Chen. Optimal parallel time bounds for the maximum clique problem on intervals. *Information Processing Letters*, 42:197–201, 1992.

[17] K. W. Chong and T. W. Lam. Finding connected components in $o(\log n \log\log n)$ time on the EREW PRAM. In *Proceedings, 4-th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–20, 1993.

[18] M. Chorbak and M. Yung. Fast algorithms for edge-coloring planar graphs. *Journal of Algorithms*, 10:35–51, 1989.

[19] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17:770–785, 1988.

[20] R. Cole and J. Hopcroft. On edge coloring bipartite graphs. *SIAM Journal of Computing*, 11:540–546, 1982.

[21] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proceedings, 27th IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[22] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[23] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal of Computing*, 15:87–97, 1986.

[24] S. A. Cook and P. McKenzie. Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8:385–394, 1987.

[25] R. Cypher and J. L. C. Sanz. Cubesort: a parallel algorithm for sorting n data items with s-sorters. *Journal of Algorithms*, 13:211–234, 1992.

[26] S. Fiorini and R. J. Wilson. *Edge-Colourings of Graphs.* Pitman, London, 1977.

[27] M. Fürer and B. Raghavachari. Parallel edge coloring approximation. *Parallel Processing Letters*, 6(3):321–329, 1996.

[28] H. N. Gabow and O. Kariv. Algorithms for edge coloring bipartite graphs and multigraphs. *SIAM Journal of Computing*, 11:117–129, 1982.

[29] M. R. Garay, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[30] J. Gil and Y. Matias. Fast hashing on a PRAM—designing by expectation. In *Proceedings, 2nd ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1991.

[31] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proceedings, 32nd IEEE Symposium on Foundations of Computer Science*, pages 698–710, 1991.

[32] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry breaking in sparse graphs. In *Proceedings, 19th Annual ACM Symposium on Theory of Computing*, pages 315–324, 1987.

[33] M. Goldberg and T. Spencer. Constructing a maximal independent set in parallel. *SIAM Journal of Discrete Mathematics*, 2(3):322–328, 1989.

[34] M. Goldberg and T. Spencer. A new parallel algorithm for the maximal independant set problem. *SIAM Journal of Computing*, 18(2):419–427, 1989.

[35] T. Goldberg and U. Zwick. Optimal deterministic approximate parallel prefix sums and their applications. In *Proceedings, 3rd Israel Symposium on Theory of Computing and Systems*, pages 220–228, 1995.

[36] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Computer Science and Applied Mathematics, Academic Press, New York, 1980.

[37] M. T. Goodrich, Y. Matias, and U. Vishkin. Optimal parallel approximation for prefix sums and integer sorting. In *Proceedings, 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 241–250, 1994.

[38] V. Grolmusz and P. Ragde. Incomparability in parallel computation. *Discrete Applied Mathematics*, 29:63–78, 1990.

[39] T. Hagerup. Self-simulation on the PRAM. Unpublished Manuscript, 1990.

[40] T. Hagerup. Fast deterministic processor allocation. *Journal of Algorithms*, 18:629–649, 1995.

[41] T. Hagerup. The parallel complexity of integer prefix summation. *Information Processing Letters*, 56:59–64, 1995.

[42] T. Hagerup, M. Chrobak, and K. Diks. Optimal parallel 5-colouring of planar planar graphs. *SIAM Journal of Computing*, 18(2):288–300, 1989.

[43] T. Hagerup and M. Kutylowski. Fast integer merging on the EREW PRAM. *Algorithmica*, 17:55–66, 1997.

[44] T. Hagerup and R. Raman. Waste makes haste: Tight bounds for loose parallel sorting. In *Proceedings, 33rd IEEE Symposium on Foundations of Computer Science*, pages 628–637, 1992.

[45] T. Hagerup and R. Raman. Fast deterministic approximate and exact parallel sorting. In *Proceedings, 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–10, 1993.

[46] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, 1989.

[47] P. Hajnal and E. Szemerédi. Brooks colouring in parallel. *SIAM Journal of Discrete Mathematics*, 3(1):74–80, 1990.

[48] Y. Han. Matching partition a linked list and its optimization. In *Proceedings, 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 246–253, 1989.

[49] Y. Han. Parallel algorithms for linked lists and beyond. In *Proceedings, International Symposium SIGAL 90*, pages 86–100. Lecture Notes in Computer Science, 450, Springer-Verlag, 1990.

[50] F. Harary. *Graph Theory.* Addison-Wesley, 1969.

[51] J. P. Hayes. *Computer Architecture and Organization.* McGraw Hill, 1988.

[52] I. J. Holyer. The NP completeness of edge coloring. *SIAM Journal of Computing*, 10:718–720, 1981.

[53] N. Immerman. Expressibility and parallel complexity. *SIAM Journal of Computing*, 18(3):625–638, 1989.

[54] K. Iwama and Y. Kambayashi. A simpler parallel algorithm for graph connectivity. *Journal of Algorithms*, 16:190–217, 1994.

[55] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[56] T. R. Jensen and B. Toft. *Graph Coloring Problems.* John Wiley & Sons, Inc., 1995.

[57] M. Karchmer and J. Naor. A fast parallel algorithm to colour a graph with δ colours. *Journal of Algorithms*, 9:83–91, 1988.

[58] H. J. Karloff. *Fast parallel algorithms for graph theoretic problems: matching, coloring and partitioning.* PhD thesis, University of California, Berkeley, 1985.

[59] H. J. Karloff. An NC algorithm for Brook's theorem. *Theoretical Computer Science*, 68(1):89–103, 1989.

[60] H. J. Karloff and D. B. Shmoys. Efficient parallel algorithms for edge coloring problems. *Journal of Algorithms*, 8:39–52, 1987.

[61] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[62] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of ACM*, 32(4):762–773, 1985.

[63] P. N. Klein. Efficient parallel algorithms for chordal graphs. *SIAM Journal of Computing*, 25(4):797–827, 1996.

[64] D. Knuth. *Sorting and Searching*. Addison-Wesley, 1973.

[65] C. E. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, C-32:181–185, 1983.

[66] L. Kučera. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14:93–96, 1982. also 17, (1983), p107.

[67] D. Lee and E. Batcher. A multiway merging network. In *Proceedings, 5th International Symposium on Algorithms and Computation, ISAAC*, pages 643–651. Lecture Notes in Computer Science, 834, Springer-Verlag, 1994.

[68] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, c-31(4):344–354, 1982.

[69] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, c-30(2):93–100, 1981.

[70] W. Liang. Private communication, 1997.

[71] W. Liang, X. Shen, and Q. Hu. Parallel algorithms for the edge-coloring and edge-coloring update problems. *Journal of Parallel and Distributed Computing*, 32:66–73, 1996.

[72] N. Linial. Distributive graph algorithms—global solutions from local data. In *Proceedings, 28th IEEE Symposium on Foundations of Computer Science*, pages 331–336, 1987.

[73] L. Lovasz. *Combinatorial Problems and Exercises*. North-Holland, Amsterdam, 1979.

[74] M. Luby. A simple parallel algorithm for the maximal independant set problem. *SIAM Journal of Computing*, 15(4):1036–1053, 1986.

[75] M. Luby. Removing randomness in parallel computation without a processor penalty. *Journal of Computer and System Sciences*, 47:250–286, 1993.

[76] J. Misra and D. Gries. A constructive proof for Vizing's theorem. *Information Processing Letters*, 41:131–133, 1992.

[77] D. E. Muller and F. P. Preparata. Bounds to complexities of networks for sorting and switching. *Journal of ACM*, 22:195–201, 1975.

[78] O.Berkman and U. Vishkin. On parallel integer merging. *Information and Computation*, 106:266–285, 1993.

[79] O.Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal of Computing*, 22:221–242, 1993.

[80] A. Panconesi and A. Srinivasan. The local nature of δ-coloring and its algorithmic applications. *Combinatorica*, 15(2):255–280, 1995.

[81] I. Parberry. *Parallel Complexity Theory.* John Wiley & Sons, 1987.

[82] N. Pippenger. Communication networks. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier Science Publishers, 1990.

[83] P. Ragde. The parallel simplicity of compaction and chaining. *Journal of Algorithms*, 14(3):371–380, May 1986.

[84] P. Rajčáni. Optimal parallel 3-colouring algorithm for rooted trees and its applications. *Information Processing Letters*, 41:153–156, 1992.

[85] G. Sajith. Parallel RAM algorithms for colouring graphs. M. Tech. Thesis, Indian Institute of Technology Kanpur, 1992.

[86] G. Sajith and S. Saxena. Parallel edge colouring of $\log^{O(1)} n$ degree graphs. In *Proceedings, 3rd (Indian) National Seminar on Theoeretical Computer Science*, pages 292–298, 1993.

[87] G. Sajith and S. Saxena. Optimal parallel algorithms for coloring bounded degree graphs and finding maximal independent set in rooted trees. *Information Processing Letters*, 49:303–308, 1994.

[88] G. Sajith and S. Saxena. Optimal parallel algorithm for brooks' colouring bounded degree graphs in logarithmic time on erew pram. *Discrete Applied Mathematics*, 64:249–265, 1996.

[89] J. E. Savage and M. G. Wloka. A parallel algorithm for channel routing. In J. van Leeuwen, editor, *Graph Theoretic Concepts in Computer Science, Lecture Notes in Computer Science 344*, pages 288–301. Springer-Verlag, 1988.

[90] S. Saxena. Two-coloring linked lists is NC^1-complete for logarithmic space. *Information Processing Letters*, 49:73–76, 1994.

[91] S. Saxena, P. C. P. Bhatt, and V. C. Prasad. On parallel prefix computation. *Parallel Processing Letters*, 4:429–436, 1994.

[92] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computational model. *Journal of Algorithms*, 2:88–102, 1981.

[93] M. Snir. On parallel searching. *SIAM Journal of Computing*, 14:688–708, 1985.

[94] R. Sridhar and N. Chandrashekharan. Highly parallelizable problems on sorted intervals. *Parallel Computing*, 21:433–446, 1995.

[95] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14:862–874, 1985.

[96] L. G. Valiant. Parallelism in comparison models. *SIAM Journal of Computing*, 4:348–355, 1975.

[97] Z. Wen. Multiway merging in parallel. *IEEE Transactions on Parallel and Distributed Computing*, 7:11–17, 1996.

[98] A. Wigderson. The complexity of graph connectivity. In *Proceedings, 17th International Symposium on Mathematical Foundations of Computer Science*, pages 112–132. Lecture Notes in Computer Science, 629, Springer-Verlag, 1992.

[99] R. J. Wilson. *Introduction to Graph Theory*. Longman, London, 1979.

CSE-1997-D-SAJ-PAR